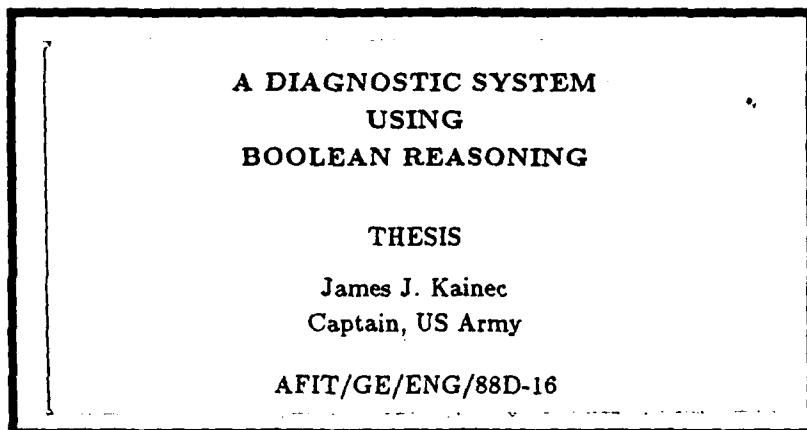


DTIC FILE COPY

1

AD-A202 764



DTIC
ELECTE
S JAN 18 1989
CD

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

89 1 17 149

(1)

AFIT/GE/ENG/88D-16

DTIC
ELECTRONIC
S JAN 18 1989
D 28 D

A DIAGNOSTIC SYSTEM
USING
BOOLEAN REASONING

THESIS

James J. Kainec
Captain, US Army

AFIT/GE/ENG/88D-16



Accession For	
NTIS	GT-881
DTIC	188
Update	1
Julian	1
By	
Dist	1
A-1	

Approved for public release; distribution unlimited

AFIT/GE/ENG/88D-16

A DIAGNOSTIC SYSTEM USING BOOLEAN REASONING

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

James J. Kainec, B.S.

Captain, US Army

December, 1988

Approved for public release; distribution unlimited

Acknowledgments

I would like to express my appreciation to my wife [REDACTED] for her love and patience which allowed me to devote the effort required to conduct this research.

I wish to thank Doctor Frank Brown for his valuable guidance while allowing me the greatest freedom possible throughout this endeavor. I called upon his wealth of knowledge in all phases of this effort. I also wish to thank Dr. Brown, Major Joseph DeGroat, and Doctor Henry Potoczny for reviewing this thesis and adding their constructive comments.

James J. Kainec

Table of Contents

	Page
Acknowledgments	ii
Table of Contents	iii
List of Figures	ix
List of Tables	xii
Abstract	xiii
I. Introduction	1
Background	1
Problem	6
Scope	7
Assumptions	7
Standards	8
Methodology	8
Overview	10
II. Summary of Current Knowledge	12
Introduction	12
Boolean-Based Methods	14
Use of Line-Condition Equations	14
Boolean Difference	19
Prime Implicants	20
Boolean Equations and Path Sensitization	23
Other Boolean Methods	28
Path Sensitization Approaches	30

	Page
The D Algorithm	31
The PODEM Algorithm	35
The FAN Algorithm	37
Line-Deduction Methods	38
The Deduction Algorithm	38
The Elimination Algorithm	40
The GEMINI System	41
Diagnosis of Non-Classical Faults	42
Bridging Fault Detection	42
Stuck-Open and Stuck-On Fault Detection	44
Abstract-Level Approaches	45
Artificial Intelligence Techniques	47
Diagnosis Based on Structure and Behavior	47
The DART Program	48
The General Diagnostic Engine	50
Summary	51
III. Problem Analysis	52
Introduction	52
Model-Based Diagnosis	53
The Choice of Model and Reasoning Method	55
The Implementation Language	61
Conclusion	64
IV. Mathematical Development of the Diagnostic System	65
Revision of the Checkpoint Model for Stuck-at Faults	65
Derivation of a Characteristic Equation	69
Generation of Effective Test Vectors	77

	Page
Deduction of New Information	80
Interpretation of Results	87
Derivation of the Circuit Function	88
Determination of Possible Checkpoint States	90
Applications of the Diagnostic Algorithm	92
Advantages and Limitations of the Diagnostic Algorithm	94
 V. Architecture of the Diagnostic System	 97
The Input Module	98
Boolean Equation Format	98
Definition of VHDL Input Format	103
The Equation Generation Module	109
The Tester Module	112
The Interpretation Module	115
Extensions to the Architecture	120
 VI. Implementation of the Diagnostic System	 122
Introduction	122
A Diagnostic Session	129
The Input Module	129
Boolean Equation Input	133
VHDL Input	136
VHDL Description Validation	138
The Equation Generation Module	138
Generation of Input and Output Lists	138
The Use of the $f = 0$ Form	139
The Choice of Data Structure	140
Formation of the Characteristic Equation	141

	Page
The Tester Module	145
Test Generation	146
Formation of a New Constraint	148
The Interpretation Module	149
Determination of the Actual Circuit Function	149
Determination of the Designed Circuit Function	150
Comparison of the Actual to the Designed Function	151
Deduction of Fault Conditions	151
Summary of Portability Issues	154
Optional Procedures	155
MS DOS Interfacing Procedures	157
Scheme Extensions	157
Summary	158
VII. Results	160
Algorithm Complexity	160
Circuit Diagnosis	164
Detection of Redundancy	165
System Limitations	170
VIII. Conclusions and Recommendations	171
Summary	171
Assessment	172
Conclusions	173
Recommendations	174
Improvement of Existing System	175
Extension of the Model and Algorithm	177
New Models and Reasoning Methods	178

	Page
Appendix A: Terminology in Fault Diagnosis	180
Appendix B: Fundamentals of Boolean Algebra	187
Definitions	187
Axioms	188
The Inclusion Relation	189
Theorems	189
Properties	191
Literals, Terms, and Formulas	191
Functions	192
Boolean Expansion Theorem	194
Extended Verification Theorem	194
Canonical Forms	194
Minterm Canonical Form	195
Maxterm Canonical Form	196
Blake Canonical Form	196
Reduction	198
Eliminants	201
The Conjunctive Eliminant	201
The Disjunctive Eliminant	202
Elimination	202
Solutions of Boolean Equations	203
Comparison of Functions	204
Appendix C: Definition of VHDL Subset	205
BNF Notation	207
Entity Declarations	208
Architecture Declarations	209

	Page
Concurrent Statements	210
Expressions	211
Port Interface List/Declarations	212
Identifiers	213
Appendix D: Diagnostic System Code	214
Bibliography	348
Vita	353

List of Figures

Figure	Page
2.1. Example of Circuit Checkpoints	17
2.2. Example of a Two-Level AND-OR Network	21
2.3. Example of Paige's Method	22
2.4. Example of Path Sensitization	24
2.5. Example of Reconvergent Fanout	26
2.6. Circuit with Numbered Checkpoints	27
2.7. Circuit Illustrating Derivation of Characteristic Function	29
2.8. Circuit Illustrating Test Vector Generation with the D Algorithm	33
2.9. Result of a Bridging Fault	43
3.1. Circuit for Example 3.1	56
4.1. Checkpoint Logic Gate	66
4.2. Karnaugh Map for the Checkpoint Logic Gate	67
4.3. Alternate View of a Checkpoint Logic Gate	68
4.4. Circuit Used in Equation Derivation	69
4.5. Circuit with Checkpoint Logic Gates Inserted	70
4.6. Circuit for Example 4.1	74
5.1. The Input Module	99
5.2. Circuit Demonstrating Boolean Equation Format	100
5.3. Equations Representing Circuit of Figure 5.2	103
5.4. VHDL Entity Declaration of the Circuit in Figure 5.2	106
5.5. VHDL Architecture Body of the Circuit in Figure 5.2	109
5.6. The Equation Generation Module	110
5.7. The Tester Module	113

Figure	Page
5.8. Input Equation and Test Vector for the Circuit in Figure 5.2	114
5.9. The Interpretation Module	116
5.10. Designed and Actual Circuit Functions for B s-a-0 in Figure 5.2	117
5.11. Output of Information that is Certain About Node States	118
5.12. Fanout Nodes and Associated Logic Gates	119
5.13. Output of Possible Fault Cases	120
5.14. Example Performance Metric Output	121
6.1. Evaluation of a Scheme Primitive Procedure	123
6.2. Format for the Definition of a User-Defined Procedure	124
6.3. An Example of a User-Defined Procedure Definition	124
6.4. Defining and Using a New Procedure	125
6.5. Example of the Main Procedure Definition	126
6.6. A Diagnostic Session (Screen 1)	130
6.7. A Diagnostic Session (Screen 2)	131
6.8. A Diagnostic Session (Screen 3)	132
6.9. A Diagnostic Session (Screen 4)	133
6.10. List of Characters (Boolean Equation Representation)	133
6.11. List of Symbols (Boolean Equation Representation)	134
6.12. Revised List of Symbols (Boolean Equation Representation)	134
6.13. Intermediate Format	135
6.14. Intermediate Format (Suffixes Added)	135
6.15. List of Characters (VHDL Representation)	136
6.16. List of Symbols (VHDL Representation)	137
6.17. First Revised List of Symbols (VHDL Representation)	137
6.18. Second Revised List of Symbols (VHDL Representation)	137
6.19. Intermediate Format (Unique Fanout Branch Labels)	142
6.20. Sum-of-Products Representation	142

Figure	Page
6.21. Sum-of-Products Representation (Revised Input Node Labels)	142
6.22. Sum-of-Products Representation (With Checkpoint Equation Added)	143
6.23. Sum-of-Products Representation (After Elimination of BX-)	144
6.24. Sum-of-Products Representation (After Insertion of Checkpoint Equations)	144
6.25. Sum-of-Products Representation (After Elimination of Internal Node Variables)	145
6.26. Characteristic Equation	146
6.27. An Updated Characteristic Equation	146
6.28. The Final Characteristic Equation	150
6.29. The Exclusive-OR of the Circuit Inputs and Output	150
6.30. Sum-of-Products Form of the Circuit Under Diagnosis	151
6.31. Sum-of-Products Form After Elimination of Internal Nodes	151
6.32. The Checkpoint State Equation	152
6.33. The Complemented Form of the Checkpoint State Equation	152
6.34. The Complemented Checkpoint State Equation with Common Literals Removed	154
 7.1. Diagnosis of Circuit with XOR Gate and Two Faults	165
7.2. Diagnosis of Circuit in Figure 7.1	166
7.3. Diagnosis of Circuit in Figure 7.1 (cont.)	167
7.4. A Redundant Circuit	167
7.5. Diagnosis of a Redundant Circuit	168
7.6. Diagnosis of a Redundant Circuit (cont.)	169

List of Tables

Table	Page
2.1. Evaluation Criteria for Diagnostic Systems	13
2.2. Singular Cover for a Two-Input AND Gate	32
2.3. Singular Cover for Two-Input AND Gate with Input s-a-0	32
2.4. Primitive D-Cube for Two-Input AND Gate with Input s-a-0	32
2.5. Propagation D-Cube for Two-Input OR Gate	33
2.6. Initial Node Assignment Using the Primitive D-Cube	33
2.7. Node Assignment Using a Propagation D-Cube	34
2.8. Final Node Assignment Using the D Algorithm	35
3.1. Results of Input-Output Experiment	58
3.2. Results of New Test	60
4.1. Truth Table for the Checkpoint Logic Gate	67
5.1. VHDL Operators and Their Boolean Equation Analogs	107
6.1. Processing Times to Form Single Equation	140
7.1. Comparison of Actual Number of Prime Implicants to the Theoretical Case	163
B.1. Truth Table for Example B.1	193
B.2. Shorthand Notation for Minterms	195
B.3. Shorthand Notation for Maxterms	197
B.4. Results of Example B.9	204

Computer aided

Abstract

The goal of this thesis is to design and implement a diagnostic system for combinational circuits, a type of circuit used in the design of all computers. The diagnostic system is to accept a description of the circuit, supervise an adaptive input-output experiment on a potentially faulty implementation of the circuit, and return the locations of all faults in the circuit.

The description of the circuit which will be input to this system can be in one of two forms: Boolean equations, or statements in the VHSIC Hardware Description Language (VHDL). Based on the circuit structure, a fault model is developed which can be used to mathematically model the state of faults in the circuit. The circuit description is processed to derive a single Boolean characteristic equation; information gained from testing is used to update this equation. The characteristic equation is manipulated to generate test vectors which are used as inputs to the actual circuit being diagnosed. After a given test vector has been input to the circuit, the output is observed. The state of the circuit output is then input to the diagnostic system which uses it to derive new knowledge about the actual circuit. Such tests are conducted repetitively until the diagnostic system determines that further information cannot be derived from testing. At this point, the diagnostic system determines the nature and location of faults in the actual circuit as well as the function actually performed by the circuit. (KR : ←)

The basis of the proposed system is Boolean algebraic reasoning. Such reasoning is performed in terms of symbols rather than numbers; thus a conventional computer language is unsuited for this application. Symbolic programming languages such as LISP or PROLOG, however, handle the manipulation of symbols extremely well. The new diagnostic system is implemented in SCHEME, a modern, efficient dialect of the LISP programming language.

A DIAGNOSTIC SYSTEM USING BOOLEAN REASONING

I. Introduction

Background

As computers have permeated society, the need for reliable computer systems has grown. It is now commonplace for systems to be used in situations where accurate real-time processing of data is required. However, the growing complexity of integrated circuits, particularly Very Large Scale Integration (VLSI) class chips, has made the design and manufacture of reliable systems increasingly difficult. Consequently, the testing phase of the design process has grown in importance. Before a system is used, it must be tested to reasonably insure it is error-free. If defects are detected, localization of the errors may be required either to correct the system under test or to build future versions correctly. In digital systems design and testing, *fault diagnosis*¹ encompasses the detection and location of defects in a system [Lala 85:25]. Thus, fault diagnosis is now one of the most important aspects of the design process.

A *fault*, or *physical fault*, is defined as any physical discrepancy in a circuit [Fujiw 85:8]. *Fault detection* is the determination that faults exist in a circuit; *fault location* is the localization of faults in a circuit [Abram 80:452]. Before a digital system is fielded, it should be ascertained that the system produces the correct outputs for a set of expected inputs. An array of signals simultaneously applied to all circuit inputs is called a *test vector*. The application of a single test vector to a circuit is called a *test*. In fault detection, an *experiment* is conducted in which a *test set*, a set of test vectors, is applied to a circuit and the reaction of the circuit to the inputs is observed [Rai 87:263]. The test outputs are then compared to a pre-computed set of expected outputs. If the two sets of outputs do not match, then one or more faults were detected in the circuit. Because

¹The definition of all italicized words is in Appendix A: Terminology in Fault Diagnosis.

it is normally too costly to exhaustively test a circuit with all conceivable input combinations, only a representative set of the possible inputs is used. Ideally, the number of test vectors in a test set should be minimal, i.e., the set of test vectors should be reduced to the smallest number that will detect all detectable faults in the circuit. Note that a distinction is made between all faults and all *detectable* faults. Certain faults called *redundant* faults cannot be detected because a circuit performs correctly even when they occur. *Irredundant* faults are faults which are not redundant.

A test set is generated using a *fault model* of a circuit. A fault model represents the types of faults that may occur in a circuit. Faults are classified as either *logical* or *non-logical*. Logical faults cause "the logical function of a circuit element (or elements) or an input signal to be changed to some other function" [Breue 76b:15]. All faults that are not logical faults are called non-logical or *parametric* faults. Examples of logical faults are *stuck-at*, *bridging*, *stuck-open* and *stuck-on* faults. Stuck-at faults are faults in which a line is constantly at a high voltage level (*stuck-at-1*) or at a low level (*stuck-at-0*) thus causing the line to remain at a fixed logical value [Al-Ar 87b:360]. Because most logical faults are modeled successfully with stuck-at faults, the stuck-at fault model is the most widely used—hence stuck-at faults are often called *classical* faults. Bridging faults are faults in which two lines are shorted together. Stuck-open faults, unique to the MOS technology used in integrated circuits, make a *combinational* circuit element into a *sequential* circuit element² [Al-Ar 87b, Lala 85, Fujiw 85]. Stuck-open faults are caused by a transistor being stuck in an open state.³ Stuck-on faults are caused by a MOS transistor being stuck in a closed state. Non-logical faults include defects due to circuit timing and power anomalies [Lala 85, Fujiw 85]. Physical faults are generally modeled as logical faults. Breuer states:

By modeling physical faults as logical faults, the problem of fault analysis becomes a logical problem which is frequently technology independent in the sense that the same model is applicable to many technologies. In addition, tests derived for logical faults may be useful for physical faults whose effect on circuit behavior is incompletely understood or too complex to be analyzed. [Breue 76b:18-19]

²Combinational circuits are those whose outputs are dependent solely on their current input values; sequential circuits are circuits with some form of memory.

³MOS transistors are switches which are either open (off) or closed (on).

Faults may also be classified as either *permanent* or *temporary*. Permanent faults, sometimes called *solid* faults, do not change with time [Nagle 75:484]. Temporary faults are divided into two types: *transient* and *intermittent*. Intermittent faults appear on some regular basis, while transient faults appear arbitrarily [Lala 85:20]. Temporary faults are difficult to detect or isolate because they may disappear when a circuit is tested. Temporary faults only can be modeled using probabilistic methods [Lala 85:20]. Since probabilistic data are not usually available, permanent faults are generally assumed in a logical fault model [Breue 76b:19]. Thus, the predominant fault model is one based on a permanent logical fault.

Once a fault model has been chosen for a circuit, a test set may be derived. Typically, a description of the circuit structure is used in conjunction with the fault model to algorithmically generate a test set that will detect faults represented by the model. For example, a test set may detect all stuck-at faults in a circuit. Sometimes different faults may have the same effect on the input-output behavior of a circuit; these faults are called *equivalent* faults. A set of equivalent faults is called an *equivalence class* [McClu 71:1286]. A test which detects one fault in an equivalence class detects all faults in the equivalence class. Thus, a test set need only detect one fault from each equivalence class [McClu 71:1286]. A common metric for judging the utility of a test set is the percentage of the total number of faults in the circuit that the test set may detect. This percentage is called the *fault coverage* of a test [Fujiw 85:17].

Although more computationally complex than fault detection, fault location is important because the localization of faults may be required for replacement of defective components or to insure that future versions of a circuit are built correctly [Fujiw 85:12-14]. In fault location, the input-output experiment may be either *adaptive* or *preset* [Breue 76a, Abram 80]. Lala defines these terms:

In "adaptive" experiments the choice of the input symbols is based on the output symbols produced by a machine earlier in the experiment. In "preset" experiments the entire input sequence is completely specified in advance. [Lala 85:51]

In preset experiments, a *fault dictionary*, a table in which the specific faults detected by a test set are enumerated, is used to determine the location of faults. A fault dictionary is produced by determining the circuit response to each of the vectors in the test set for every possible fault condition [Nagle 75:503]. Once a circuit is fabricated, its response to the test set for which the dictionary was developed is compared to the entries of the dictionary to determine which faults were detected by the test set. Because of the many different fault cases that have to be considered for each test vector, fault location using preset input-output experiments is usually restricted to a single-fault assumption. A circuit with k lines has at most $2k$ possible single stuck-at faults. If n is the number of test vectors in a test set, then the number of calculations necessary to create a fault dictionary would be $2kn$. However, the same circuit has $3^k - 1$ possible multiple faults. Thus, $(3^k - 1)n$ would be the number of calculations required to generate a fault dictionary for the multiple fault case. Because the dictionary grows exponentially with the number of faults considered, it is impractical to generate a fault dictionary for the multiple fault case [Abram 80:452]. Clearly, a preset fault-location scheme requires more computation than does a preset scheme for fault detection. Before a fault dictionary is even constructed for a preset fault-location procedure, the test vectors for which it will be devised must be created by an algorithm that generates a detection test set.

The basis for a fault location algorithm using an adaptive input-output experiment is a model of the system to be tested. The algorithm uses this representation to derive an initial test vector for the circuit. The outputs of the circuit are then observed. The reaction of the circuit to the input is used to derive information necessary to generate a subsequent test vector. Additionally, the results of each iteration of the experiment are used to progressively determine the location of detectable faults in the circuit, if any exist. The test proceeds until further information cannot be gained and/or test vectors cannot be generated. Information derived in the adaptive experiment is often used to determine the function realized by the faulty circuit. Thus, the result of the experiment is an indication of the location of faults in the circuit, if any exist and are detectable, and possibly the

function the circuit is performing [Abram 80, Breue 76a, Solan 86]. In general, adaptive diagnostic systems that currently exist are not genuinely adaptive. These systems require the generation of a detection test set prior to operation. Based on the results of a diagnostic experiment using this test set, new test vectors are generated to gain the information required to isolate existing faults. Thus, while most adaptive systems eliminate the need for a fault dictionary, the requirement for generation of a detection test set remains. Adaptive diagnostic systems may be used for fault detection, however more typical is that they are used for fault location. Unless otherwise noted, it is assumed in this project that adaptive systems are used to locate faults.

Most fault-diagnostic systems that have been implemented assume that a faulty circuit contains a single stuck-at fault. The three most widely known algorithms for fault detection, the D algorithm, PODEM, and FAN, are all based on the single fault assumption [Kirk 88:43]. Preset fault location systems assume a single fault to limit the size of the fault dictionary. Adaptive fault location systems may isolate multiple faults; however, they are still dependent on test generation systems which generate test sets based on a single fault assumption. Although research continues on the use of diagnostic systems assuming a single fault [Hughe 86, Agarw 81], multiple-fault diagnostic systems are a necessity due to the complexity of Very Large Scale Integration (VLSI) circuits [Abram 80, Jacob 87]. VLSI devices contain more than 5000 gates [Johns 87:72]. Some reasons for diagnosing multiple faults include:

1. They occur often in new devices.
2. Single nonclassical faults can be detected by test sets generated by algorithms based on a multiple stuck-at fault assumption.
3. The set of single faults that can occur in a circuit is just a subset of the set of possible multiple faults. [Breue 76a:44]

Lala states that "statistical studies have shown that multiple faults, composed of at least six single faults, must be tested in an LSI chip to establish its reliability" [Lala 85:47]. Clearly, as integrated circuit chips get denser, the multiple-fault case cannot be ignored.

Fault diagnosis algorithms are inherently complex even when a single fault is assumed. Generating a test to detect a single stuck-at fault in a combinational circuit is an NP-complete problem [Abadi 85:9]. Nevertheless, the complexity of the problem does not diminish the need for a solution. Rather, innovative approaches are required.

Problem

Currently-available diagnostic systems are based on restrictions or assumptions which limit their capability. Such systems typically assume that only a single fault may occur in a circuit, perform fault detection but not fault location, or are preset rather than adaptive. The objective of this thesis is to design and implement a diagnostic system which can adaptively locate multiple faults in combinational circuits. The system will accept a description of a circuit, supervise an input-output experiment on a potentially faulty implementation of the circuit, and return the location of all faults detected in the circuit.

The circuit description may be in the form of Boolean equations or a VHSIC Hardware Description Language (VHDL) description. The system will guide a user through the testing of the actual circuit. After a test vector is generated by the diagnostic system, it will be input to the circuit. The state of the circuit outputs will then be input to the diagnostic system. Each test vector generated should be *effective*, i.e., the resulting circuit output should not have been deducible from the results of prior tests. Hence, the system will yield a near-optimal test sequence. Once the diagnostic system determines that further information cannot be derived through the input-output experiment, the location of faults, to within irreducible equivalence classes, and the identification of the actual circuit function will be output by the diagnostic system.

Scope

To keep the scope of the project manageable, the following restrictions will be imposed:

- The type of circuit which will be diagnosed by this system will be a single-output combinational logic circuit. Diagnosis of multiple-output circuits is an extension of the single-output case; the system can be extended in the future to handle multiple-output circuits. Sequential circuit diagnosis is an extension to combinational circuit diagnosis.
- A subset of VHDL will be used to describe the circuit. This subset will be clearly specified.⁴
- A symbolic language will be used to rapidly prototype the system. This will allow concentration on the problem and not on the programming. Efficiency issues are subordinate to attaining a working system.
- Although ideas may be elaborated concerning potential uses for an adaptive diagnostic system, the only aspect which will be implemented in this project is a system which will adaptively locate all multiple faults to within equivalence classes. Other uses, e.g., generation of tests to detect a specific fault, are variations of locating all multiple faults.

Assumptions

The fault model to be used in this project is the stuck-at fault model. There has been dispute about the validity of this model for CMOS technology [Baner 84, Bate 88, Burge 88]. However, contrary evidence has been found that concludes that using the stuck-at fault model for CMOS device testing has not degraded product quality [Turne 85]. Other research supports the conclusion that a high percentage of errors will be detected and located using the stuck-at model.

It will be assumed that the user has access only to the *primary inputs* and *primary outputs* of the circuit. Primary inputs are "the externally accessible input pins of a circuit through which we can inject logical values into the circuit" [Kirk 88:48]. Primary outputs are "the externally accessible output pins of the circuit through which we can observe logical values from the circuit" [Kirk 88:48]. This means that a user cannot probe internal points in the circuit. Given this limitation, the best possible fault resolution is to locate irredundant faults to within equivalence classes [McClu 71:1286]. In the case of this project, multiple faults will be located to within equivalence classes.

⁴See Appendix C: Definition of VHDL Subset

Other assumptions important to this project are that testing does not affect the potentially faulty circuit in any way, and that when a fault occurs it becomes permanent. These assumptions are used in most diagnostic systems [deKle 87:100]. Moreover, logical fault models such as the stuck-at model normally assume the permanence of faults [Breue 76b:19].

Standards

This research is a proof of concept. The goal of this project is to develop a diagnostic system that adaptively locates multiple faults. Efficiency and optimization of system design are not the goals of this research; the requirement is to develop a functional system.

Methodology

The first step in developing the diagnostic system was to analyze other approaches to fault detection and location. This analysis was important in gaining insight into how to design the system. Methods were classified by type. Several techniques provide a basis for the development of a diagnostic system to locate multiple faults.

After other techniques were examined, the approach taken in this project was determined. A system based on Boolean reasoning was chosen. This choice was based on several factors. A substantial amount of research has been done previously in developing the use of Boolean equations in fault diagnostic systems which detect and locate multiple faults [Bosse 71, Breue 76a]. Thus, this work extends previous techniques. Furthermore, Boolean expressions are easily manipulated using symbolic programming languages. A considerable number of Boolean problem-solving techniques have been implemented in the Scheme⁵ symbolic programming language [Brown 88b]. These procedures were used in this work. The use of a symbolic language enabled the rapid prototyping of the system; this allowed concentration on the problem and not the programming. Furthermore, versions of Scheme are available for both personal computers and minicomputers.

⁵The Scheme language is a dialect of LISP.

Once the diagnostic method was chosen, an algorithm to generate test vectors, perform reasoning, and interpret results was developed. All tests generated by the algorithm are effective. Information gained from the results of previous tests guide the test generation process. This algorithm is the basis for the diagnostic system.

The next stage in the problem solving process was the development of an architecture for the system which included user interfaces, partitioning of the system, and the determination of data to be passed between the partitions. The input formats chosen to describe a circuit were Boolean equations or a VHDL description. To limit the scope of the project, a subset of VHDL was specified for use. The output format chosen was a listing of the possible faults in the circuit and an equation representing the function that the circuit is performing. The system architecture was partitioned into four components. The components are the input module, equation generation module, tester module, and interpretation module. The input module converts either of the two input formats to a single intermediate representation of the circuit. The equation generation module converts the intermediate representation into a single Boolean equation. The test module uses this Boolean equation in conducting an input-output experiment. The interpretation module uses the output of the tester module, a Boolean equation, to generate the fault listing and circuit-function equation.

The final step in the process was to implement the system. A PC-based version of Scheme was used to construct the system. However, care was taken to conform with the Revised³ Report on the Algorithmic Language Scheme, the document upon which all implementations of the Scheme language are based [Rees 86]. Deviations from the report were documented. Example circuits were tested to determine the capabilities of the system.

Overview

This chapter has provided the background of this project as well as a problem definition. The scope of the effort as well as assumptions were presented. A general approach to the solution of the problem was outlined.

Criteria for examining diagnostic systems are developed in Chapter 2. Current diagnostic systems are discussed with respect to these criteria. Methods are classified by approach. Restrictions and limitations of these systems are discussed. Different approaches to the diagnostic problem include Boolean-based techniques, path sensitization approaches, line-deduction methods, abstract-level approaches, systems developed to diagnose non-classical faults, and methods which use artificial intelligence techniques. Adaptive diagnostic systems have been developed using line-deduction and artificial intelligence approaches.

Chapter 3 is a detailed problem analysis. The attributes of an ideal diagnostic system are examined. The necessity of a model and reasoning method for adaptive diagnosis is discussed. The choice of the cause-effect equation for the circuit model and Boolean reasoning for the reasoning method are developed. The symbolic programming language Scheme is to be used as the implementation language for the system. The choice of Scheme vis-à-vis other languages is examined.

In Chapter 4, the mathematical development of the approach taken in this diagnostic system is shown. Derivations are presented to establish how a characteristic equation is developed to model the state of the circuit, how test vectors are generated, how information is gained from an input-output experiment, and how results are interpreted. Appendix B. Fundamentals of Boolean Algebra, gives an overview of the concepts of Boolean algebra that are used in this chapter.

The architecture of the diagnostic system developed in this project is discussed in Chapter 5. Items discussed in this chapter include: functional decomposition of the system into modules; descriptions of the operations performed in each module; data to be transferred between modules; and user interfaces. Descriptions of the input circuit format using either Boolean equations or

VHDL are given. Appendix C, Definition of VHDL Subset, defines the subset of VHDL that may be used to describe a circuit to be diagnosed by this system.

Chapter 6 describes the implementation of the diagnostic system. Detailed descriptions of design trade-offs are discussed. Language portability issues also are examined. An example diagnostic session is used as a reference point for discussing the system implementation.

The results of this project are presented in Chapter 7. Issues which are examined include a discussion of the complexity of the diagnostic algorithm developed in Chapter 4, VHDL subset validation, capabilities of the system to diagnose faults and detect redundancy, and limitations of the system that was developed. Examples are given which demonstrate the use of the diagnostic system.

Chapter 8 presents a summary of the effort. An assessment is made of the strengths and limitations of the diagnostic system. The conclusions of this effort are stated. Recommendations for future work are elaborated.

II. Summary of Current Knowledge

A voluminous amount of research has been conducted in the area of fault diagnosis. As digital system designs become more complex there will be a continual need for increasingly sophisticated diagnosis algorithms that match this complexity. Furthermore, as technology changes, fault models that previously sufficed may account for only a portion of the errors that may occur in new technologies. Thus, new models and algorithms must be developed to cope with these advances. A survey of approaches taken previously is helpful in gaining an insight into tasks involved in developing new methods.

Introduction

Diagnostic systems can be evaluated in terms of the functions they perform. A system may generate only a fault detection test set or may be used for fault location; it may assume a single fault or may handle the multiple-fault case; it may be adaptive or preset. Other criteria, outlined below, are also of importance.

Some diagnostic algorithms place restrictions on the structure of the circuit to be diagnosed, while other methods can be used to diagnose any type of logic circuit. This distinction is important because when restrictions are placed on the structure of the circuit the latitude of the circuit designer becomes limited.

A priori enumeration of faults is required in some methods, but not required in others. Enumeration of faults is impractical in any situation other than when the single-fault assumption is used. As noted earlier, a circuit with k lines has $2k$ possible single stuck-at faults, but $3^k - 1$ possible multiple faults.

It may be important to generate tests to diagnose faults other than classical stuck-at faults. This may be of particular importance in cases where the probability of non-classical faults, such as

stuck-open or bridging faults, is high. Most techniques assume classical faults, while other methods can detect and/or locate non-classical faults.

Finally, in some diagnostic systems a transformation of a representation of the circuit is required, whereas other systems require no such transformation. A *transformation* of a circuit representation involves a conversion from the actual view of the circuit to an equivalent form, e.g., the conversion from the actual transistor representation of the circuit to a gate-level description. Two circuits are said to be *equivalent* if their input-output behavior is indistinguishable. The diagnostic system then operates on the equivalent view; tests generated for the latter view detect faults which occur in the actual representation of the circuit.

All of these criteria are given in Table 2.1. Current diagnostic systems will be compared with respect to these criteria. In some cases systems which take a very different approach to the diagnostic problem are very similar in relation to these measures, whereas in other instances the differences are substantial. Most all of the existent diagnostic methods assume that faults are permanent and logical.

Evaluation Criteria	
Limited to Detection	• Can Locate Faults
Single Fault Assumption	• Multiple Fault Assumption
Preset Test Vector Generation	• Adaptive Test Vector Generation
Restrictions on Circuit Structure	• No Restrictions on Structure
Fault Enumeration Required	• No Enumeration Required
Classical Faults Assumed	• Non-classical Faults Diagnosed
Transform of Circuit Representation	• No Transformation Required

Table 2.1. Evaluation Criteria for Diagnostic Systems

Different approaches to the diagnostic problem include systems based on some form of Boolean manipulation, path sensitization approaches, line-deduction methods, systems developed specifically to diagnose non-classical faults, abstract-level approaches, and techniques which have a foundation in Artificial Intelligence. The remainder of this chapter examines these approaches by giving

examples of each and discussing systems with respect to the evaluation criteria previously developed.

Boolean-Based Methods

Many diagnostic systems use some form of Boolean¹ manipulation. However, even these approaches differ vastly from one another. A survey of differing techniques includes those which use line-condition equations to generate test vectors, approaches which use the concept of the Boolean difference, two which use the prime implicants of a circuit equation to generate test vectors, those which combine the use of Boolean equations and path sensitization, and a collection of other disparate techniques.

Use of Line-Condition Equations. Diagnostic procedures based on line-condition equations use Boolean equations to represent the circuit under diagnosis. Included in this equation are variables representing the state of internal lines in a circuit. Diagnosis is performed by generating tests which yield information regarding the state of these variables. Methods which use line-condition equations include Poage's method, Bossen and Hong's procedures, and Breuer, Chang, and Su's on-line method for fault location. [Breue 76b:35]

Poage's Method. Perhaps the most significant effort in the use of Boolean methods for fault diagnoses was the technique proposed by Poage in his landmark paper on the detection of faults in combinational circuits [Poage 63]. Using the stuck-at model, he developed a single equation in which an output signal of a circuit was a function of both the input variables and the condition of every line in the circuit. Each line in the circuit has three variables associated with it.

¹It may be helpful to scan Appendix B, Fundamentals of Boolean Algebra, before proceeding.

For example, for a line a :

- $a_n = 1$ if line a is normal, 0 otherwise,
- $a_0 = 1$ if line a is stuck-at-0, 0 otherwise,
- $a_1 = 1$ if line a is stuck-at-1, 0 otherwise. [Poage 63:487]

Note that exactly one of the three variables will take the value of 1 at any given time, because the conditions of normal, stuck-at-0² and stuck-at-1 are mutually exclusive and collectively exhaustive. These variables are inserted into the original circuit equation to form a new output equation to model the circuit. The idea is that if any wire would become permanently stuck-at-0 or stuck-at-1, the variables with respect to that wire would be changed accordingly; the new output equation would parallel the actual function of the faulty circuit. If all lines in the circuit are normal, then all a_n variables would be set to 1, and the output equation would reflect the function of the circuit as originally designed. Poage showed how to use the derived circuit equation to generate a minimal test set to detect all single stuck-at faults. Each single fault has to be enumerated in the process [Poage 63:501-504]. Using the concept of *fault collapsing* he also developed a procedure to generate a minimal test set to detect multiple faults without having to enumerate all possible multiple faults. In fault collapsing, faults are combined based on a study of the circuit to be diagnosed [Scher 72:859]. Thus, tests are developed for classes of collapsed faults rather than individual fault cases. Poage's method uses a preset test vector generation scheme, places no restrictions on the circuit structure, and does not perform any transformation of the circuit representation.

The primary criticism of Poage's method is that when the circuit is very large, the number of variables required is so great that the technique becomes computationally infeasible.

²Henceforth, the shorthand notation of s-a-0 will denote stuck-at-0, s-a-1 will denote stuck-at-1.

Bossen and Hong's Procedures. Bossen and Hong modified Poage's method to develop an algorithm to generate a test set to detect all multiple stuck-at faults in a combinational network. The importance of their work is that they developed the concept of using only specific points, called *checkpoints*, in a circuit to generate test vectors.

The set of faults that have the same effect on the network output behavior are [sic] called equivalent and any test pattern that detects any one of them also detects equivalent faults . . . it becomes possible to represent all the distinctive faults by specifying a minimal number of points in the network such that any multiple faults in the network become equivalent to a multiple faults [sic] among these checkpoints. [Bosse 71:1252]

Two types of lines are checkpoints in a network: all primary inputs which do not fan out, and all fanout branches. A *fanout* is a point at which several lines are connected. One of the lines in the fanout is the originating line, called the *fanout stem*. The other lines, which are beyond the fanout point, are called *fanout branches*. See Figure 2.1 for an example. Based on this concept, only the checkpoints are used to develop test vectors, because multiple faults at the checkpoints are equivalent to multiple faults at arbitrary nodes in the circuit. Bossen and Hong modified Poage's single circuit equation so that a circuit output is a function of the input variables as well as the state of the checkpoints. The variables that Poage developed to represent the state of each line in the circuit were used by Bossen and Hong to represent the condition of each checkpoint [Bosse 71:1253]. These variables are called the *checkpoint variables*. They named their version of the single circuit equation the "cause-effect equation" [Bosse 71:1253].

Using the cause-effect equation, Bossen and Hong developed a test vector generation procedure which could be used for any type of combinational circuit, and also devised a method to generate a near-minimal test set for a *nonredundant* circuit [Bosse 71]. A *redundant* circuit is one in which there exists a line which could be cut without changing the function implemented by the circuit [Nagle 75:497]. A nonredundant circuit is one which is not redundant. Redundant faults are so named because they occur in the lines that could be cut and thus do not change the function of the circuit. Different methods are often required to handle redundant networks because

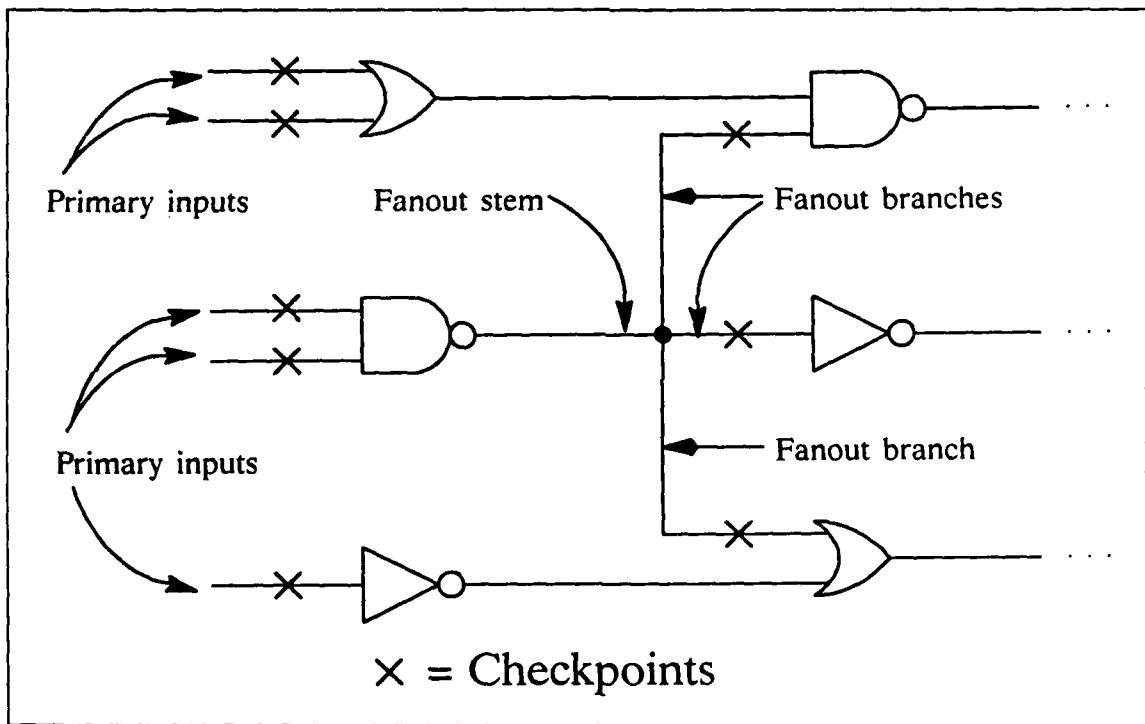


Figure 2.1. Example of Circuit Checkpoints

in redundant networks, "it is not possible to verify that every checkpoint is in the normal state" [Bossen 71:1254]. Between both of Bossen and Hong's procedures, a test set can be developed for any combinational circuit; however, their procedures can only be used for preset test vector generation. Neither fault enumeration nor a transform of the circuit representation is required using Bossen and Hong's methods.

An On-Line Procedure to Locate Faults. Breuer, Chang, and Su extended Bossen and Hong's concepts to devise what they termed an "on-line" method for locating multiple faults in a combinational circuit [Breue 76a:44]. They showed how the cause-effect equation could be used to locate faults to within an equivalence class. Their procedure requires the initial generation of a fault-detection test set by a method such as Bossen and Hong's. An experiment is conducted to determine the circuit's response to the detection test set. Then the cause-effect equation, the test vectors, and the circuit's response to the test vectors are used to produce a system of Boolean equations which are solved to obtain possible states of the checkpoint variables. These solutions are used to generate an additional test vector, the circuit's response to which is used to gain more information about the state of the checkpoint variables. This process iterates until the possible states of the checkpoint variables are determined. Note that the faults represented by the checkpoints may not indicate the actual location of a physical fault, but rather the location to within an equivalence class [Breue 76a:46]. The results of their procedure are the location of faults to within an equivalence class as well as an equation representing the faulty circuit's function [Breue 76a:44]. Breuer, Chang, and Su's procedure is pseudoadaptive: a detection test set must be generated a priori, the circuit's response determined, and then additional test vectors generated on-line.

Breuer, Chang, and Su have shown that their procedure also can be used to construct a fault dictionary for preset fault location, to identify redundancy in combinational circuits, and to determine the faults that are not detected by an arbitrary test set [Breue 76a:50]. With respect

to other evaluation criteria listed in Table 2.1, their procedure is similar to Bossen and Hong's method.

Boolean Difference. A number of methods use the Boolean difference to generate test vectors. The Boolean difference uses the XOR operation between two Boolean expressions to generate a test vector to detect a specific fault. The Boolean difference $\frac{df}{dx_i}$ of a function $f(x_1, \dots, x_i, \dots, x_n)$ with respect to a variable x_i is defined as follows:

$$\frac{df}{dx_i} = f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n) \oplus f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) \quad (2.1)$$

Given this definition, the expression $x_i \cdot \frac{df}{dx_i}$ defines the set of all test vectors which detect a s-a-0 condition on line x_i . The expression $x'_i \cdot \frac{df}{dx_i}$ defines the set of all test vectors which detect a s-a-1 condition on line x_i [Breue 76b, Fujiw 85]. In general, the formulas used in the Boolean difference method are with respect to the input signals of the circuit. Hence, all faults may be assumed to be with respect to faulty input lines [Lala 85:28]. However, the Boolean difference method has been extended to generate test vectors to detect faults on the internal nodes of a circuit [Lala 85:30].

The Boolean difference method is typically used to generate test vectors to detect all single stuck-at faults in a circuit. The reason for this is that all faults must be listed a priori and a separate Boolean difference expression must be generated for each fault case. A fault dictionary can be generated using Boolean differences for preset fault location. Bearson and Carroll have used the Boolean difference method to generate a minimal length test set to detect single faults in irredundant combinational circuits [Bearn 71]. Additionally, several researchers have developed methods for multiple fault detection using Boolean differences. Ku and Masson are normally credited with this accomplishment [Fujiw 85, Rai 78]. Yau and Tang developed a method which combined the Boolean difference with path sensitization³ techniques to generate test vectors to detect multiple faults [Yau 71]. Rai and Aggarwal devised another method which simplified the calculation of Boolean differences for generation of test vectors for multiple-fault detection. Carroll,

³Path sensitization is discussed in a later section.

Shah, and Jones developed an expression they called the "generalized test function," the basis for which is the Boolean difference. The generalized test function is used to generate all test vectors to detect a given multiple stuck-at fault in an irredundant combinational circuit [Carro 74].

All methods based on the Boolean difference have the primary weakness that faults must be enumerated to facilitate test vector generation. The only restriction on circuit structure is that in several approaches, the circuit must be irredundant. However, none requires a masking analysis or transformation of the circuit representation.

Prime Implicants.

Paige's Method. The objective of Paige's method is to generate test sets to detect single stuck-at faults in irredundant combinational networks with a minimum of analysis [Paige 69:2]. Paige developed a procedure to generate test sets to detect faults in the gates of two-level AND-OR networks (Figure 2.2). An irredundant two-level AND-OR network implements a subset of the prime implicants of a function. He demonstrated that a fault can be detected by determining whether any of the prime implicants or prime implicants change to become independent of any variables [Paige 69:8]. To detect this circumstance, Paige developed test vectors to establish whether there is an increase in the number of minterms (s-a-1 case) or maxterms (s-a-0 case), respectively, in the circuit output function [Paige 69:8]. For example, if one of the prime implicants of a function were the minterm xyz , this term would be implemented by an AND gate (Figure 2.3). If a s-a-1 fault were to occur on the z input of the AND gate, the gate then would implement the term xy . This new term is equivalent to a combination of the minterms xyz and xyz' . For this fault, the circuit output becomes independent of the z variable if $x = 1$ and $y = 1$; the minterm xyz' is added to the circuit function. Stuck-at-0 faults are handled in an analogous manner. By analysis of the faults that can occur on the inputs of the AND gates, test vectors are generated to detect the growth of minterms or maxterms.

Paige extended his method to generate test vectors for multi-level networks. However, these networks have restrictions on the types of gates that can be used. The reason for this restriction is that the circuit representation is transformed into an equivalent AND-OR network to perform test vector generation.

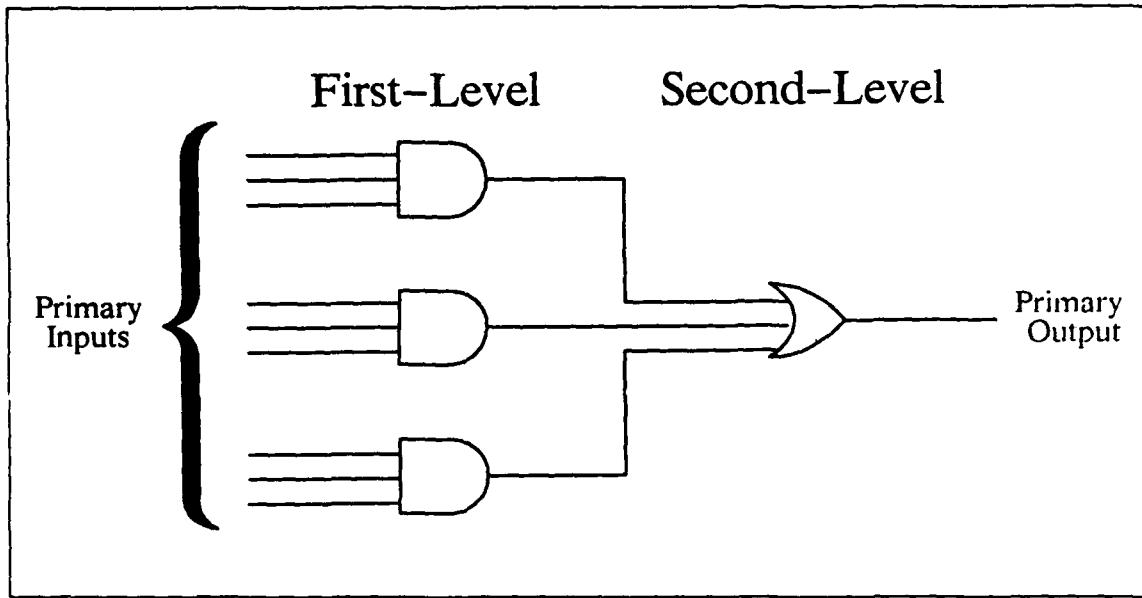


Figure 2.2. Example of a Two-Level AND-OR Network

Kohavi's Procedures. Kohavi and Kohavi developed procedures very similar to Paige's approach to generate test sets for stuck-at fault detection in multi-level combinational circuits [Kohav 72]. The rationale for test vector selection is essentially the same as Paige's. However, they give two procedures for use in test vector generation. They show how to use a Karnaugh map to generate test vectors for circuits with a small number of variables, and also give a general algorithm which can be used for any number of variables.

Two transformations of the circuit representation are required prior to the generation of test vectors. First, a network must be transformed into an equivalent fanout-free circuit. A *fanout-free* network is one which has no fanout. Then, the fanout-free representation is transformed into

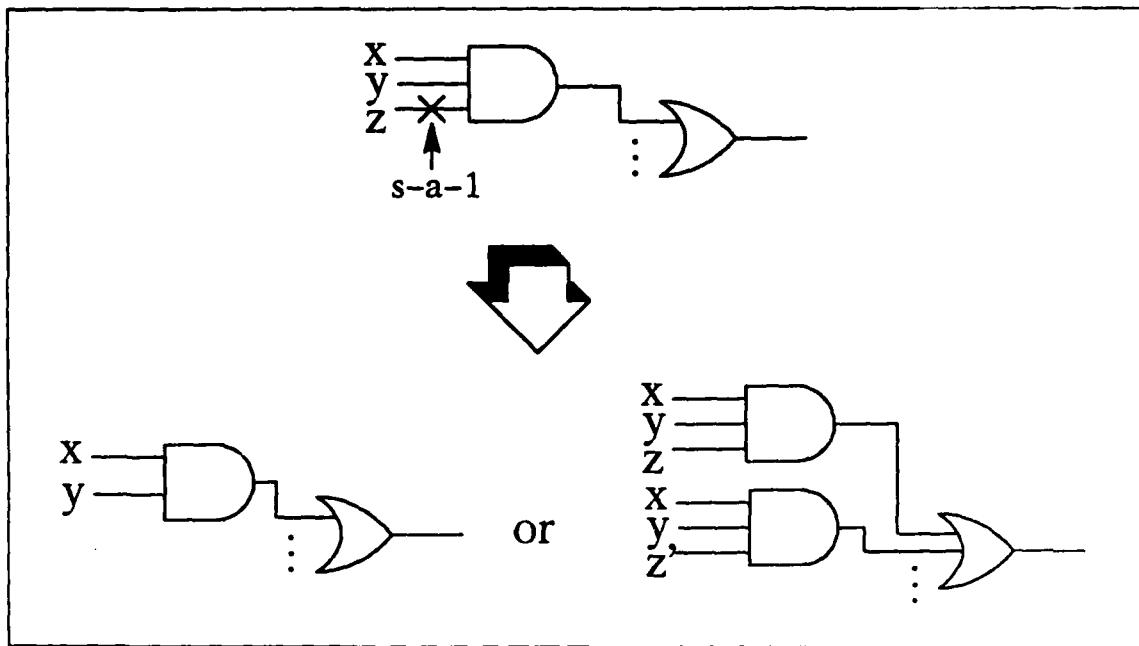


Figure 2.3. Example of Paige's Method

an equivalent AND-OR network such that the Boolean formula which represents the circuit is an irredundant sum of prime implicants [Kohav 72:558]. After the transformations, a test set is generated for single-fault detection in the equivalent AND-OR network. Kohavi and Kohavi show that for two-level networks, such as the AND-OR network, a test set that detects all single faults also detects all multiple faults [Kohav 72:563]. Test sets that detect all faults in the equivalent AND-OR network detect all faults in the original circuit.

When the original circuit is a two-level AND-OR network, Kohavi's procedures yield a minimal test set; a near-minimal test set is generated for most multi-level circuits. Some restrictions are placed on circuit redundancy by these procedures. As in Paige's procedure, the experiments produced are preset.

Boolean Equations and Path Sensitization.

Equivalent Normal Form. Armstrong combined the use of a Boolean expression to represent a circuit with the concept of path sensitization. Path sensitization is the establishment of circuit conditions such that a hypothetical fault condition is propagated from the point of the fault to a primary output resulting in an output different than what would occur in a fault-free circuit. Causing this condition is called "sensitizing the path" [Armst 66:67]. Given the circuit in Figure 2.4, to detect a stuck-at-1 condition on line a , a path is traced from line a to the primary output z . For all of the AND and NAND gates in the path, the remaining inputs on the gates must be set to 1, while the remaining inputs of all OR and NOR gates must be set to 0 [Armst 66:67].

The primary inputs for the given test are chosen based on three criteria:

1. The line on which the hypothetical fault exists must be set to the opposite value from the fault for which the test is being devised. If the line for which the test is devised is several levels into the circuit, then the primary inputs which can affect the line would have to be chosen accordingly.
2. Primary inputs which affect the remaining inputs of the gates in the sensitized path are set to insure that the remaining inputs of AND and NAND gates are set to 1, OR and NOR gates are set to 0.
3. Primary inputs which do not have any effect on the sensitized path are chosen arbitrarily.

In conformance with these rules, three alternative test vectors will detect a stuck-at-1 condition on line a in Figure 2.4. These are:

- $a = 0, b = 1, c = 0, d = 0, e = 0$
- $a = 0, b = 1, c = 0, d = 1, e = 0$
- $a = 0, b = 1, c = 1, d = 0, e = 0$

To generate test vectors using path sensitization, Armstrong used a Boolean expression which he called the "equivalent normal form" [Armst 66:68]. The equivalent normal form is nothing more than a sum-of-products Boolean formula for the function that the circuit implements. However, subscripts are added to each literal to denote the path through the circuit, and redundant terms and literals are not deleted from the derived formula [Armst 66:67-68]. The subscripts denote gates

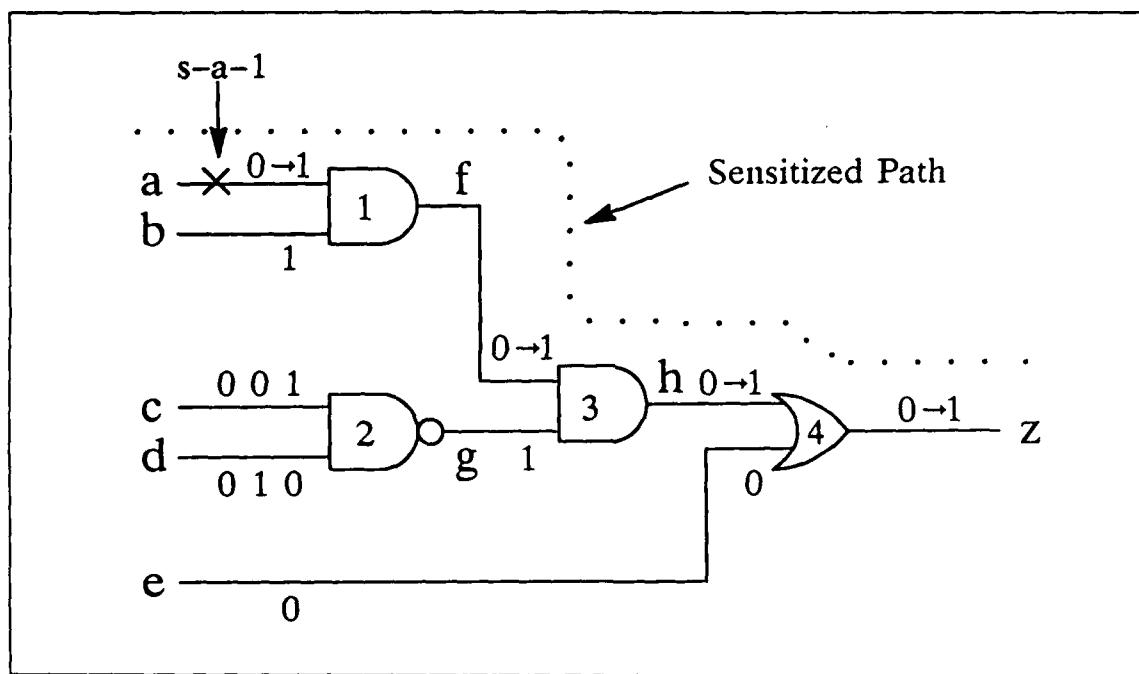


Figure 2.4. Example of Path Sensitization

that an input signal passes through as it goes from primary input to primary output. Each gate is assigned a unique number. A separate formula is derived for each primary output. For the circuit in Figure 2.4, the equivalent normal form is derived as follows:

$$\begin{aligned}
 z &= (h + e)_4 & (2.2) \\
 &= ((fg)_3 + e)_4 \\
 &= (((ab)_1(cd)'_2)_3 + e)_4 \\
 &= (((ab)_1(c' + d')_2)_3 + e)_4 \\
 &= ((a_{13}b_{13}c'_{23} + a_{13}b_{13}d'_{23}) + e)_4 \\
 &= a_{134}b_{134}c'_{234} + a_{134}b_{134}d'_{234} + e_4
 \end{aligned}$$

The last equation is the equivalent normal form for the circuit in Figure 2.4.

Armstrong's algorithm uses the equivalent normal form as the basis for generating tests which detect all single stuck-at faults in a combinational circuit. Prior to test vector generation, the circuit representation must be converted to equivalent normal form. Within the procedure, a test is developed to detect a fault on one of the input lines. This test also detects selected faults along an associated sensitized path. Test vectors are generated until it is determined that faults on all paths are covered. A problem with the algorithm is that the rule regarding how to assign the remaining inputs of gates in a sensitized path, i.e., 1 for AND and NAND gates, 0 for OR and NOR gates, does not hold for some circuits with reconvergent fanout (Figure 2.5) [Armst 66:67]. *Reconvergent fanout* is a condition such that at least two fanout branches from the same fanout form paths which come together at the inputs to one or more gates between the fanout and a primary output. The algorithm may not generate test vectors to detect all faults that can occur on a reconvergent fanout path. Thus, test vectors cannot be generated for some combinational circuits

with reconvergent fanout. Because only one path is sensitized to detect a fault, Armstrong's method implements what is normally called "one-dimensional path sensitization" [Lala 85:27].

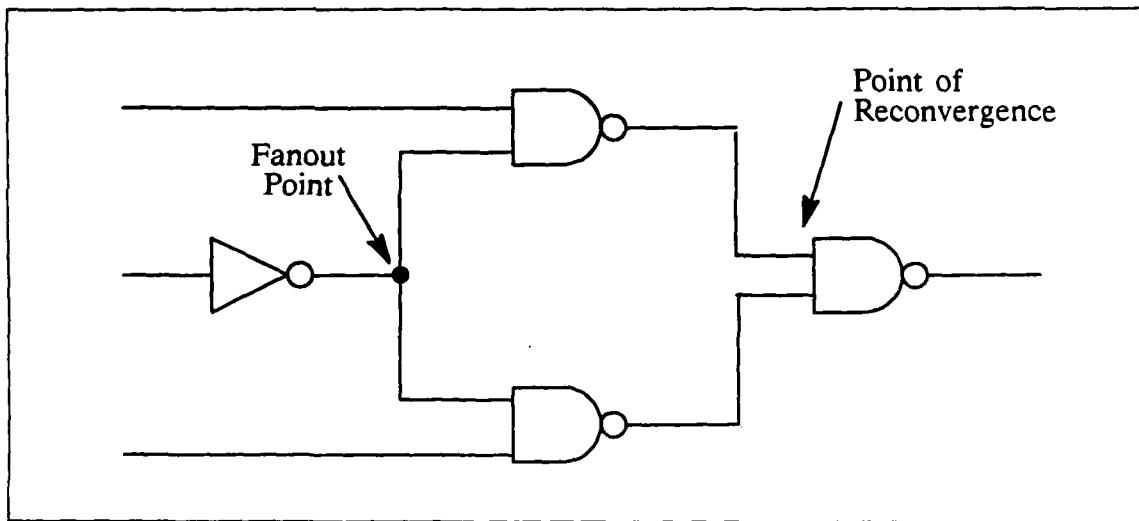


Figure 2.5. Example of Reconvergent Fanout

Equivalent Sum-of-Products. Fridrich and Davis developed a method to generate minimal test sets to detect all single faults and near-minimal test sets to detect all multiple faults in an irredundant combinational circuit. They borrowed the concept of checkpoints from Bossen and Hong and the equivalent normal form from Armstrong to form what they called the "equivalent sum-of-products" form [Fridr 74:850]. While Armstrong numbered the gates of the circuit, Fridrich and Davis numbered the checkpoints. Thus, the equivalent sum-of-products form is the sum-of-products Boolean formula for the function that the circuit implements—with subscripts denoting the path through the circuit via checkpoints [Fridr 74:851]. Similar to Armstrong, redundant terms and literals are not deleted from the derived formula [Fridr 74:850]. Prior to test-set generation,

the circuit representation is transformed to equivalent sum-of-products form. The following equation represents the equivalent sum-of-products form for the circuit in Figure 2.6:

$$f = a_{15}b_{25}c'_{37} + a_{15}b_{25}d'_{47} + a'_{16}c_{23}d_{48} + b'_{26}c_{38}d_{48} \quad (2.3)$$

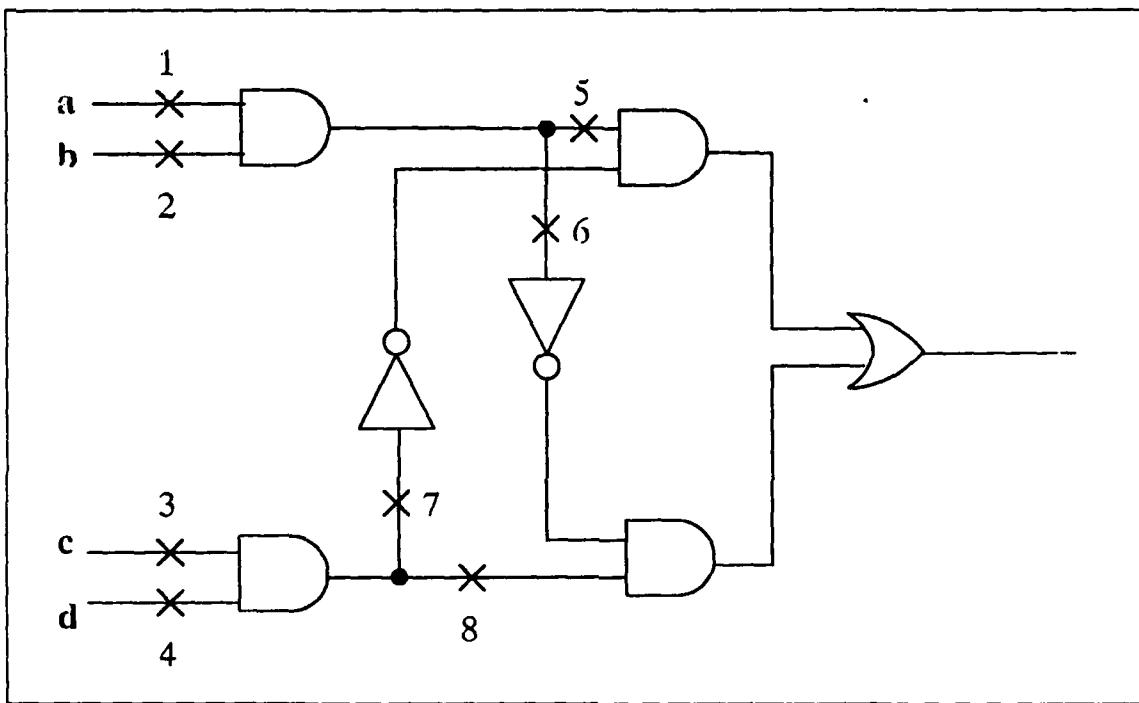


Figure 2.6. Circuit with Numbered Checkpoints

Using the equivalent sum-of-products form of the circuit, Fridrich and Davis developed a procedure which yields a minimal test set to detect all single stuck-at faults at the checkpoints. Generation of a test set which detects multiple faults is a three-step process. First, the generation of the test set for detection of single faults must be performed. Second, a masking analysis must be performed to determine the multiple faults at the checkpoints that are not detected by the single-fault test set. When a fault f masks another fault f' for a given test vector, f' would normally be detected by the test vector if it occurred as a single fault, but would not be detected by the same

test vector if both f and f' occurred simultaneously [Fridr 74:855]. A *masking analysis* is required to determine all cases in which one fault masks another for a given test set. The computation required to perform this analysis may be substantial. Finally, a procedure is used to generate tests that detect the multiple faults found in the second step. The test vectors produced by the last step are combined with the single-fault test set to provide a near-minimal test set to detect all multiple faults in an irredundant circuit [Fridr 74:858].

Other Boolean Methods. There are many other methods which use some form of Boolean equations to generate test sets. Cerny employs what he calls "characteristic functions" to generate tests to detect faults at a given point in the circuit. Cha uses a concept called "prime faults" to generate test sets.

Characteristic Functions. Cerny showed how to generate test vectors to detect stuck-at and bridging faults at any point in an irredundant combinational circuit. Central to his method is the concept of characteristic functions [Cerny 76:1]. A characteristic function is a Boolean function representing an entire circuit or subcircuit in terms of the inputs and outputs.⁴ The characteristic function $\Phi(\underline{x}, z)$ is defined as the Exclusive NOR, \odot , of the inputs and the circuit output. For the circuit in Figure 2.7, the equation defining the circuit is:

$$z = x_1x_2 + x_3x_4. \quad (2.4)$$

The characteristic function of the circuit is derived as follows:

$$\begin{aligned} \Phi(\underline{x}, z) &= z \odot (x_1x_2 + x_3x_4) \\ &= z(x_1x_2 + x_3x_4) + z'(x_1x_2 + x_3x_4)' \\ &= x_1x_2z + x_3x_4z + x'_1x'_3z' + x'_1x'_4z' + x'_2x'_3z' + x'_2x'_4z' \end{aligned} \quad (2.5)$$

⁴See Equations B.46-B.54 in Appendix B.

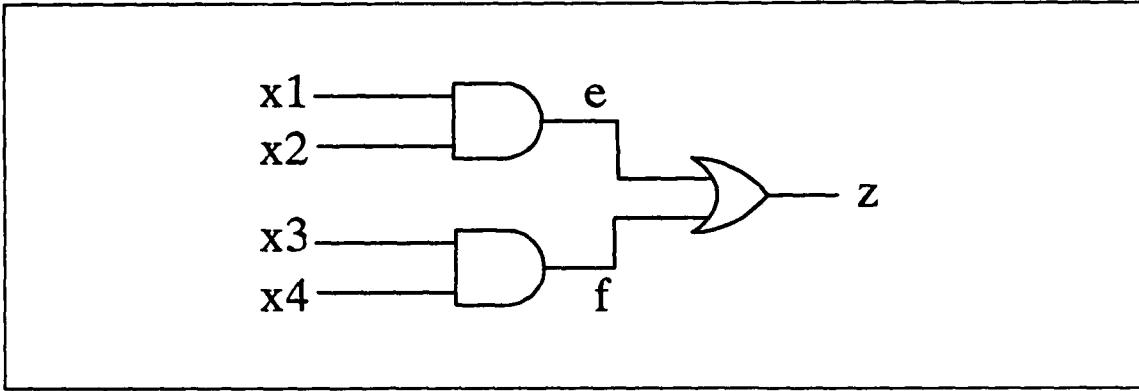


Figure 2.7. Circuit Illustrating Derivation of Characteristic Function

In Cerny's procedure, to derive a test to detect a fault at a point in a circuit, the point is treated as an output of a subcircuit. A characteristic function is then derived for this subcircuit. The arguments of the characteristic function are the primary inputs to the subcircuit as well as the output of the subcircuit. The characteristic function is then manipulated to form a set of tests to detect a fault at the output of the subcircuit [Cerny 76:10-12]. Cerny extended his procedure to detect double faults [Cerny 76:18]. The weakness of the approach is that all faults must be specified a priori to generate tests to detect them. Furthermore, a covering procedure must be used to select a minimal number of tests from the set of all tests that the procedure generates. Strengths of the procedure are that bridge faults can be modeled and no transformation of the circuit structure is required to generate tests.

Prime Faults. Cha developed a concept called "prime faults" to generate a multiple-fault detection test set for a combinational circuit. The approach is somewhat similar to Bossen and Hong's approach via checkpoints. Instead of checkpoints, the locations at which faults are modeled are called "prime fault sites". Two criteria define the selection of prime fault sites. First, all checkpoints are prime fault sites. Second, the output of a prime gate which does not fan out is a prime fault site. A prime gate is a gate in which all of its inputs are checkpoints of the circuit.

However, inverters are skipped; if an input to an inverter is a fanout branch or a primary input, then the gate connected to the output of the inverter is a prime gate. [Cha 79:149-150]

There are more prime fault sites than checkpoints in a circuit. However, prime fault sites involve only two variables, whereas checkpoints as defined by Bossen and Hong have three associated variables. Checkpoints can be normal, s-a-0, and s-a-1. Prime fault sites can be normal, and either s-a-0 or s-a-1. Only one fault condition is possible at a prime fault site [Cha 79:150]. Cha showed that using prime faults reduces the number of variables that must be manipulated in a Boolean equation. He also discovered that relations between prime faults allow the number of fault cases that must be considered when generating tests to be reduced [Cha 79:150]. Cha developed two algorithms, one for generating tests to detect a prime fault, the other for generating a multiple-fault detection test set. Both a masking analysis and path sensitization techniques are used in generating tests to detect all multiple faults. No transformations are performed in Cha's procedures.

Path Sensitization Approaches

The most prevalent algorithms used to generate test sets to detect single stuck-at faults in combinational circuits use multiple-path sensitization. Because multiple paths are sensitized, the limitations of one-dimensional path sensitization with respect to reconvergent fanout are overcome. If a test vector exists to detect a fault in a combinational circuit of arbitrary structure, multiple-path sensitization is guaranteed to find it [Lala 85:34]. This section discusses the three most widely known algorithms which use multiple-path sensitization: the D algorithm, PODEM, and FAN.

The D Algorithm. The D algorithm is considered to be "the first algorithmic method for generating tests for non-redundant combinational circuits" [Lala 85:34]. The D algorithm uses the calculus of D-cubes to generate test vectors. In this calculus, two symbols represent differences between conditions of a faulty and good circuit. These symbols and the conditions they represent are:

- D , 1 in the good circuit, 0 in the faulty circuit,
- \bar{D} , 0 in the faulty circuit, 1 in the good circuit. [Kirk 88:47]

To generate a test vector to detect a specific fault on a node in the circuit, a D or \bar{D} is assigned to the node. A D is assigned if the fault condition is stuck-at-0, a \bar{D} is assigned to designate a stuck-at-1 condition. After assignment of a D or \bar{D} to the potentially faulty node, values of 0, 1, D , and \bar{D} are assigned to the remaining nodes of the circuit. A value of D or \bar{D} only can be assigned to nodes on paths between the potentially faulty node and a primary output. The calculus of D-cubes dictates how to perform this assignment. Assignments of nodes are revised until either a D or \bar{D} is assigned to a primary output of the circuit and assignments on all nodes in the circuit are consistent. A test vector which detects the fault is an input which causes a D or \bar{D} to be propagated to a primary output of the circuit; this test vector is given by the values assigned to the primary inputs of the circuit after the node assignment process is completed. Once a D or \bar{D} is assigned to a primary output, a sensitized path is established by which the fault may be detected. The components of the calculus of D-cubes which facilitate path sensitization are the singular cover, the primitive D-cubes, and the propagation D-cubes [Lala 85:34-37].

For each gate in a circuit, a table representing the gate's singular cover is developed. The singular cover table is a condensed form of the gate's truth table [Lala 85:34]. For a two-input AND gate, the singular cover is given in Table 2.2. The symbol X represents a don't care condition.

For a gate in the circuit in which the potentially faulty node is an input or output, a second singular cover table is developed which describes the operation of the gate in lieu of the potential

x	y	$f(x, y)$
0	X	0
X	0	0
1	1	1

Table 2.2. Singular Cover for a Two-Input AND Gate

fault. Although a fault occurs logically on a node of the gate, this gate is referred to as the faulty gate. A singular cover table for a two-input AND gate in which an input node is stuck-at-0 is shown in Table 2.3.

x	y	$f(x, y)$
X	X	0

Table 2.3. Singular Cover for Two-Input AND Gate with Input s-a-0

Using the two forms of the singular cover, another table called the primitive D-cube of the faulty gate is developed. The calculus of D-cubes gives rules which dictate how to form this table. The primitive D-cube designates how to assign the values of 0, 1, D , and \overline{D} to the nodes of the faulty gate. The primitive D-cube of a two-input AND gate which has an input node s-a-0 is given in Table 2.4.

x	y	$f(x, y)$
1	1	D

Table 2.4. Primitive D-Cube for Two-Input AND Gate with Input s-a-0

For other gates in the circuit, a propagation D-cube table is developed. The propagation D-cube signifies how the values of 0, 1, D and \overline{D} are propagated from the point of the fault to a primary output of the circuit. Normal truth tables are used to assign values of nodes located between the primary inputs and the potential fault and nodes of gates which are not on the sensitized paths. A propagation D-cube table for a two-input OR gate is given in Table 2.5.

x	y	$f(x, y)$
0	D	D
D	0	D
D	0	D
0	D	D

Table 2.5. Propagation D-Cube for Two-Input OR Gate

Suppose we would like to generate a test to detect a s-a-0 condition on node a in Figure 2.8.

Initially, all nodes in the circuit would be unassigned, which is designated by a X symbol. The first step of the D algorithm is to use a primitive D-cube of the faulty AND gate to make the first node assignment in the circuit. This assignment is given in Table 2.6. After the assignment of values to nodes of the faulty gate, implication is performed. In implication, values of nodes implied by the initial assignment are determined [Kirk 88:48]. For example, once all values of input nodes of a gate are assigned, the value of the output of the gate is determined by implication. In the example circuit, no nodes are assigned using implication after the initial assignment of node values.

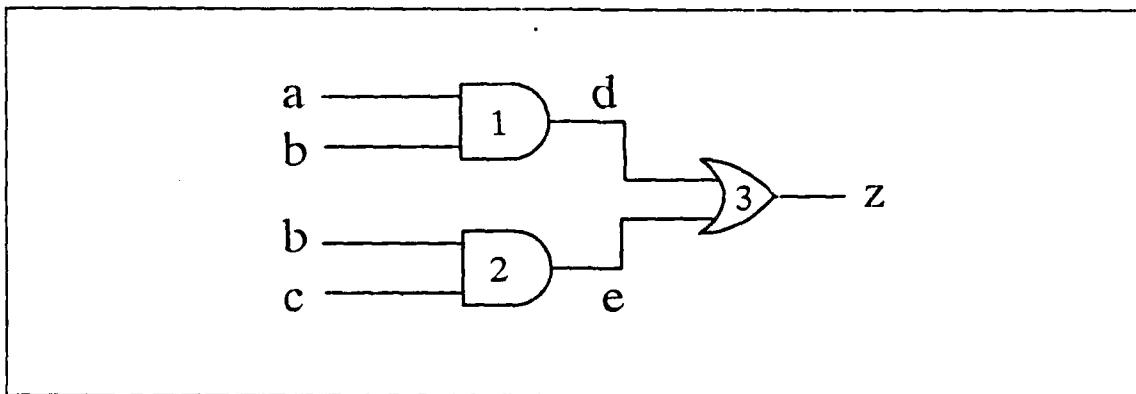


Figure 2.8. Circuit Illustrating Test Vector Generation with the D Algorithm

a	b	c	d	e	z
1	1	X	X	D	X

Table 2.6. Initial Node Assignment Using the Primitive D-Cube

The second step in the D algorithm is to assign values to a successor gate of the faulty gate using the successor gate's propagation D-cube. A successor gate is any gate which has as an input the output of the faulty gate. A propagation D-cube from a successor gate is selected which will propagate the D value. In Figure 2.8, the only successor to the faulty gate is the OR gate. Hence, a propagation D-cube for the OR gate is selected. This cube is intersected with the previous circuit node assignment shown in Table 2.7. Any value intersected with an X yields the value; all other intersections must be of like values. If a propagation D-cube cannot be intersected with the previous assignment, another propagation D-cube must be selected. This selection and intersection are shown in Table 2.7. After the assignment of values to nodes using a propagation D-cube, implication is again performed. Although not shown in the given example, a cycle of node assignment using propagation D-cubes and node implication would be performed until a D or \overline{D} results on a primary output of the circuit.

	a	b	c	d	e	z
Previous Assignment	1	1	X	D	X	X
Propagation D-Cube	X	X	X	D	0	D
Intersection	1	1	X	D	0	D

Table 2.7. Node Assignment Using a Propagation D-Cube

Once a D or \overline{D} is propagated to a primary output of the circuit, line justification is performed. Line justification is the "process of finding a set of primary input values that cause a node to take on the desired value" [Kirk 88:48]. In this example, because a value of 0 was assigned to the e node the primary input c must be chosen which will cause node e to take the value 0. The truth table for a two-input AND gate is used to determine that node c must be assigned a value of 0. With this final assignment a test is generated to detect a s-a-0 fault on node a . The final node assignment is shown in Table 2.8.

At each step in the D algorithm, assignments of node values using propagation D-cubes and truth tables must be consistent with previous assignments. Inconsistencies are found by the process of intersecting a new assignment with previous assignments as shown in Table 2.6. When

	a	b	c	d	e	z
Final Assignment	1	1	0	D	0	D

Table 2.8. Final Node Assignment Using the D Algorithm

an inconsistency is found, a backtrack is made to the last point in the node assignment process at which a choice was made. For example, when generating a test vector to detect a specific fault there may have been several gates which were successors to the gate. initially, one of these gates is chosen to propagate the D or \overline{D} using a propagation D-cube. After a number of steps in the node assignment process it may be determined that further assignments cannot be made which both propagate the D or \overline{D} to a primary output and guarantee consistency of assignments within the network. Once this determination is made, backtracking is performed to the point at which a successor gate was chosen to propagate the fault. Then, a different gate is chosen and the process of node assignment is attempted again. Essentially, a depth-first search is conducted until node assignments are made which insure both that a test vector is generated which propagates a D or \overline{D} to a primary output and that all assignment made within the circuit are consistent.

The D algorithm only can be used to generate test sets for preset fault detection experiments. The primary weakness is that each fault must be enumerated a priori. Another weakness of the D algorithm is that it is very inefficient when used to generate test vectors to detect faults in circuits which consist of many XOR gates, such as error correction and translation circuits [Lala 85:42]. Backtracking is performed often when the D algorithm is used to generate test vectors for these types of circuits. To overcome this weakness, the PODEM algorithm was developed.

The PODEM Algorithm. Like the D algorithm, the PODEM algorithm is used to generate test vectors to detect single stuck-at faults. PODEM also uses the calculus of the D-cubes to facilitate path sensitization. However, PODEM makes modifications to the D algorithm which make the search process for test vector generation more efficient [Kirkl 88].

PODEM, an acronym for Path-Oriented Decision Making, was developed to overcome the weakness of the D algorithm when generating tests for circuits with a great number of XOR gates. The point at which choices are made in the node assignment process is the primary difference between PODEM and the D algorithm. In the D algorithm, choices are made regarding the gates through which to propagate a D or \overline{D} as well as the primitive D-cubes to use in the assignment of node values to the faulty gate. After these choices, other nodes are assigned values through either implication or line justification. PODEM takes a much different approach. PODEM reduces the node assignment search space because only primary inputs must be considered when making assignments. The values of other nodes are found by implication. [Lala 85]

In PODEM, the only choices made are in the assignment of either a 0 or 1 to a primary input of the circuit. A fault is chosen for which a detection test vector will be generated. An input line is set to a value of 0 or 1 in order to cause an appropriate value to be assigned to the assumed faulty node, i.e. 0 for a s-a-1 fault, 1 for a s-a-0 fault. Implication is performed to determine all node values of the circuit which are implied by the assignment of the input node. After implication, a check is made to determine whether a D or \overline{D} is propagated to a primary output. If so, a test vector to detect the specified fault is generated; the assignment of values to the remaining unassigned primary inputs is arbitrary. If a D or \overline{D} were not propagated to a primary output, then another primary input must be assigned a value. If a conflict arises between a node value implied by the newly-assigned primary input and a node value derived from a prior implication, then backtracking occurs to the most recent input assignment. The primary input that was most-recently assigned a value then is assigned the opposite value; implication is performed. If a conflict again arises, then backtracking occurs to change the input assignment of the next-most-recently assigned primary input. Primary inputs are assigned and conflicts resolved until a D or \overline{D} is propagated to a primary output without conflicts.

The FAN Algorithm. Like PODEM, the FAN algorithm (FANout-oriented test generation) is a descendant of the D algorithm that was developed to make the test vector generation process more efficient. FAN reduces the node assignment search space by limiting choices in value assignment to specific types of nodes in a circuit, fanout stems and headlines [KirkI 88:52]. The values of other nodes are determined, through implication and justification. A headline satisfies two conditions. First, a *headline* is a line "that drives a gate that is part of a reconvergent fanout loop" [KirkI 88:52]. Additionally, a headline is an output node of a gate in which the predecessor gates are not part of any fanout loop. Due to this last property, a headline can be assigned an arbitrary value because there is no possibility of conflict when nodes between the headlines and primary inputs of the circuit are assigned values.

Values for nodes are assigned in a manner similar to the D algorithm. However, during the line justification phase of node assignment, values are assigned to nodes in a path towards the primary inputs only until a headline is reached. Thus, if conflicts were to occur at some time later in the node assignment procedure and the assignment of a headline node is changed, processing would not have been wasted determining node values between a headline and a primary input. Only after all nodes between headlines and the primary outputs of the circuit are assigned values are the nodes between headlines and the primary inputs assigned. [KirkI 88:53]

A second concept used by the FAN algorithm to improve efficiency is a method called the multiple backtrace. In the multiple backtrace, all branches of a reconvergent fanout path are examined to determine a consistent assignment of nodes prior to the actual assignment of values to any node in the reconvergent fanout path [KirkI 88:53]. Although the multiple backtrace takes processing time to perform, the benefit is that the number of backtracks in the node value assignment process is reduced. Like the D algorithm and PODEM, the FAN algorithm concludes when a D or \overline{D} is propagated to a primary output of the circuit and all nodes in the circuit are assigned without conflict.

Diagnostic systems based on the D algorithm, PODEM, and FAN are so prevalent that substantial research still is being conducted on the continued applicability of the single stuck-at fault model for fault diagnosis. Agarwal and Fung [Agarw 81:855] and Jacob and Biswas [Jacob 87:849] have performed separate analyses of the fault coverage of multiple stuck faults by single fault tests. Hughes and McCluskey have analyzed the capability of the single stuck-at model to detect multiple faults in digital circuits. They also have devised a means for adding to the test vectors determined using a single stuck-at model so that otherwise undetectable multiple faults are detected. They deem the generation of test sets based on a multiple fault model impractical due to the large number of possible stuck-at faults that can occur in a VLSI circuit [Hughe 86:368].

Line-Deduction Methods

Path sensitization methods cannot be used to locate faults in combinational circuits. To overcome this limitation, line-deduction methods have been developed which use the results of testing to deduce the states of lines within a circuit under test. The methods include Abramovici and Breuer's deduction algorithm, Solana's elimination algorithm, and Rajska and Cox's procedure.

The Deduction Algorithm. Abramovici and Breuer's deduction algorithm follows from Breuer's earlier work which used Boolean algebra to perform fault location. However, the deduction algorithm is "guided by the network topology rather than an algebraic description of the circuit" [Abram 80:452]. Abramovici and Breuer state that the use of Boolean equations is impractical "for circuits of moderate complexity" [Abram 80:452]. Their method overcomes many of the limitations of other fault diagnosis methods. Their algorithm can be used to perform multiple-fault location as well as multiple-fault detection without having to perform any form of masking analysis. A priori fault enumeration is not required. Additionally, faults other than stuck-at faults can be detected with the deduction algorithm.

The deduction algorithm shares a weakness with Breuer's on-line method for fault location. Prior to using the algorithm, a fault detection test set must be developed using some other diagnostic method. An input-output experiment then is conducted to determine a potentially faulty circuit's response to the test set. Then the test set, the circuit's response to the test set, and a description of the circuit topology are used in an effect-cause analysis to deduce the state of lines within the potentially faulty circuit. The term effect-cause is derived from the idea that the effect is viewed, then the cause is determined.

The deduction algorithm facilitates the effect-cause analysis of a circuit. This algorithm uses a test vector and the circuit's response to the test vector to perform what is called *value justification*. In value justification, an attempt is made to determine the conditions of internal nodes of the circuit which could have caused the circuit's response to the given test vector [Abram 80:454]. When performing value justification, paths between the primary inputs and the primary outputs are analyzed. Techniques used in path analysis are similar to those employed to generate test vectors using the D algorithm [Abram 80:454]. After a number of test vector/circuit response pairs are analyzed, the state of nodes in the circuit may be identified as normal, stuck-at-0, stuck-at-1, and undetermined. The best resolution of nodes states is obtained when the set of test vectors used to diagnose the circuit is a complete multiple-fault detection test set..

Even with a complete test set, however, the deduction algorithm cannot determine all of the actual line values in a circuit [Abram 80:454]. For example, only a subset of the actual normal lines in a circuit may be identified as normal. One reason for this is that a fault near an output of the circuit may block the output visibility of lines between a primary input and the fault. Additionally, the actual fault situation in a potentially faulty circuit can be determined only to within an equivalence class [Abram 80:458]. Like other fault location systems, effect-cause analysis yields a set of solutions representing the possible states of nodes within the circuit.

Abramovici and Breuer state that in some cases their algorithm may terminate without producing a set of solutions; in these cases, a fault exists in the circuit under test which is not equivalent to a stuck fault [Abram 80:454].

The Elimination Algorithm. Like Abramovici and Breuer, Solana et al. used effect-cause analysis concepts in the development of a diagnostic algorithm. Solana et al. called their method the elimination algorithm. The elimination algorithm shares many of the same characteristics with the deduction algorithm. A fault detection test set must be generated by some other test vector generation technique. A circuit's response to the test set is used to deduce the states of lines internal to the circuit. In addition, the elimination algorithm can be used for multiple-fault detection and location without a priori fault enumeration. However, a difference between the two algorithms is that the deduction algorithm views multiple-fault detection and location as inseparable, i.e. location of faults implies that they were detected, whereas the elimination algorithm can be used to perform fault detection without having to locate faults. This makes the elimination algorithm more efficient if the primary goal only is to detect faults. [Solan 86]

The elimination algorithm relies on a transformation of the circuit representation to what is called an operation map which models the behavior of the circuit. Formation of the operation map is similar to transforming the circuit representation into an equivalent fanout-free circuit [Solan 86:31]. Test vectors and the circuit's response to the test vectors are used to eliminate primary inputs from the operation map, hence the name "elimination algorithm" [Solan 86:35-36]. Implicit in the process of eliminating these inputs is the determination of the states of nodes in the circuit. After processing, nodes are classified as normal, s-a-0, s-a-1, or undetermined. As in the deduction algorithm, the elimination algorithm can be applied to an arbitrary number of test vector/circuit response pairs. However, the best line resolution is obtained when a complete fault detection test set is used. Differing from the deduction algorithm, the elimination algorithm does not have to process all test vector/circuit response pairs to determine that faults were detected in

a circuit. The elimination algorithm may be terminated as soon as it is determined that some node is faulty [Solan 86:31].

The elimination algorithm produces a set of possible node states, one of which is correct. A solution cannot be determined with certainty due to circuit redundancy, fault masking, and functional equivalence of faults. Additionally, when the elimination algorithm does not produce any set of possible solutions, the existence of a non-classical fault is indicated. [Solan 86:43]

The GEMINI System. Rajski and Cox have developed a diagnostic system of the line-deduction type that is much different from either the deduction or the elimination algorithms. In their system, they use a sixteen- value logic system called GEMINI to analyze a circuit's response to pairs of test vectors. Unlike other line deduction methods, Rajski and Cox's system generates test vectors to be input to a circuit under test. In each iteration of the diagnostic process a pair of test vectors is generated. The circuit's response to the test vector pair is analyzed to deduce information about the states of internal nodes in the circuit. [Rajsk 87]

The test vector generation process does not use information gained from the results of previous tests to generate new tests. Thus, Rajski and Cox's system does not adaptively locate faults in the conventional manner. However, because information is derived in an iterative fashion the method is not preset. A heuristic is used to generate test vectors. Each test vector pair differs in only one bit, such that a change is caused in the output of the circuit [Rajsk 87:940]. They claim that this heuristic allows information to be deduced regarding the node states in a manner more efficient than otherwise possible.

A significant aspect of their system is that it can be used to detect and locate multiple classical and non-classical faults. With their multi-valued logic system, they developed ways to implement the stuck-at, delay, and the stuck-open fault models [Rajsk 87:933-934]. These models can be used alone or in combinations when diagnosing a circuit [Rajsk 87:933]. Furthermore, any of these types

may be diagnosed using their system without the necessity of fault enumeration. Rajska and Cox's method locates faults to within equivalence classes.

Diagnosis of Non-Classical Faults

The impetus for the development of the diagnostic systems which can diagnose non-classical faults is the conclusion that the stuck-at model, whether for single or multiple faults, is inadequate for testing MOS digital circuits used in VLSI. Errors that are not modeled by the stuck-at model occur in integrated circuits. The stuck-at fault model originally was developed to detect faults in circuits composed of discrete gate-level components. However, in VLSI circuits a gate is only an abstraction representing a collection of transistors. Some faults which occur at the transistor-level are unlike stuck-at faults which occur in discrete gate-level components. One type of fault that occurs often in VLSI circuits is the stuck-open fault [Baner 84, Burge 88, Lala 85]. This type of fault is characterized by faulty operation of a MOS transistor which causes an output node to retain a prior state due to capacitive holding of electrical charge [Lala 85:18-19]. Stuck-on faults are also found in MOS circuits. Another type of fault that may occur is a bridging fault. This type of fault occurs when two wires are shorted together, often causing a change in the functionality of the circuit [Lala 85:16-18].

A number of diagnostic procedures have been developed to detect non-classical faults. Zasio suggests that methods based on these fault models should be used in addition to tests based on stuck-fault models in order to obtain sufficient fault coverage [Zasio 85:388]. These methods can be used to detect stuck-open and bridging faults.

Bridging Fault Detection. Diagnostic procedures to detect or locate bridging faults have existed for many years. Bridging faults occur in both discrete digital components as well as on integrated circuit chips. Bridging faults at the component level occur when wires or solder contact each other on circuit boards; they occur on integrated circuits due to defective manufacturing

techniques or migration of metal lines due to excess temperatures [Mei 74:720]. Most bridging faults occur at the inputs or outputs of logic gates, hence most studies of bridging faults are limited to this case. Depending on the technology of the circuit construction, a bridging fault can be viewed as the occurrence of either a wired-AND or wired-OR at the point of the fault. This is shown in Figure 2.9.

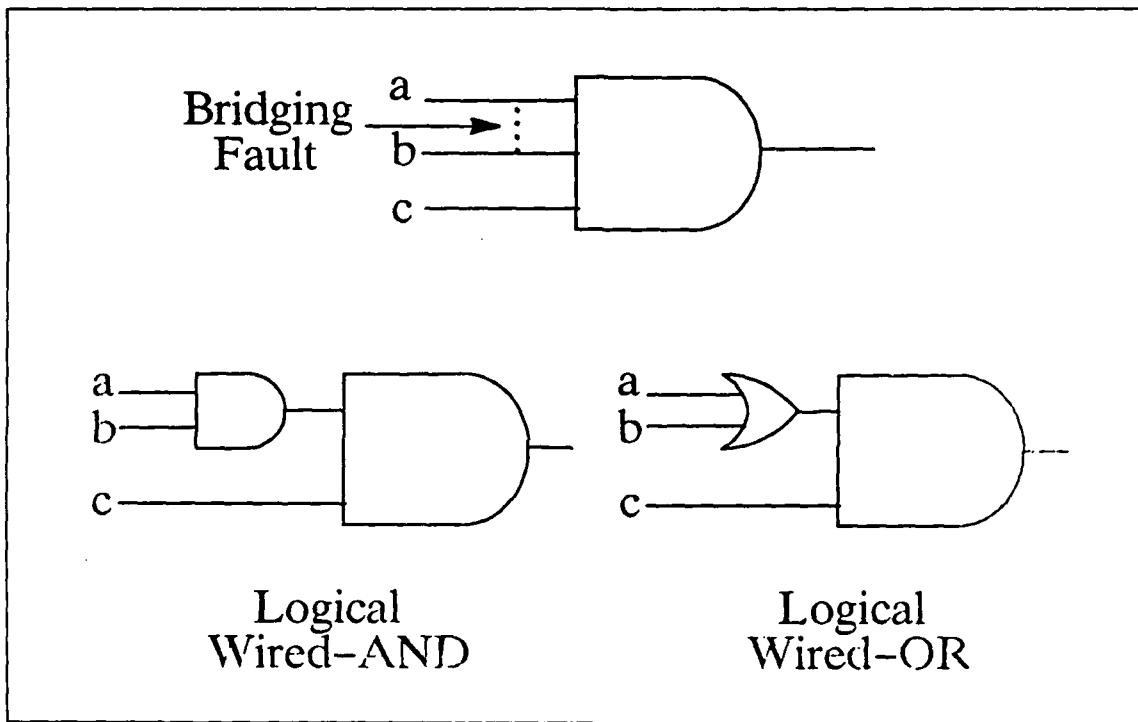


Figure 2.9. Result of a Bridging Fault

Using the bridging fault models shown in Figure 2.9, Friedman and Mei independently have shown that most bridging faults can be detected by fault detection test sets generated to detect stuck-at faults [Fried 74, Mei 74]. In both cases, they showed that the addition of constraints to a path sensitization test vector generation process will guarantee that generated test vectors will detect specific types of bridging faults. Mei also showed that the ordering of test vectors must be performed to detect certain bridging faults. Both conclude that with test vector generation

constraints and with minor limits on circuit topology, tests sets generated to detect classical faults are sufficient to detect most bridging faults [Fried 74, Mei 74].

Stuck-Open and Stuck-On Fault Detection. As with bridging faults, many methods proposed to detect stuck-open and stuck-on faults take advantage of procedures which generate tests to detect stuck-at faults. A number of these techniques accept a transistor-level description of a circuit, and transform it to an equivalent gate-level representation. Likewise, stuck-open and stuck-on faults in the transistor-level description are equivalent to conventional stuck-at faults in the gate-level representation. Procedures such as the D algorithm are used to generate a fault detection test set for the equivalent gate-level circuit. Test vectors which detect a stuck-at fault in the gate-level representation detect an analog stuck-open or stuck-on fault in the transistor-level circuit. Using circuit transformation, Al-Arian has developed procedures which can generate test vectors to detect multiple stuck-open faults in combinational circuits [Al-Ar 87b, Al-Ar 87a]. Reddy et al. [Reddy 84] and Roth [Roth 84] each have devised methods using circuit transformation which generate test vectors to detect single stuck-open and stuck-on faults in MOS combinational circuits. Jha has shown that test sets which detect the existence of single stuck-open and stuck-on faults also detect most multiple faults [Jha 86:514].

Parallel with the development of circuit transformation methods, researchers have developed procedures to detect stuck-open and stuck-on faults using transistor-level descriptions of a circuit. A common concern of these researchers is that faults modeled at the gate-level, even after transformation from a transistor-level, do not model all physical faults that can occur in a circuit. Banerjee and Abraham [Baner 84], Bate and Miller [Bate 88], and Burgess et al. [Burge 88] all have studied in depth the nature of physical faults in MOS circuits. One topic surveyed was the occurrence of faults with respect to specific MOS circuit design styles; design styles are different ways of implementing equivalent circuits. Another item of research was to determine how inadvertent node

capacitances cause faults in a circuit. Using the results of their studies, each has proposed techniques for test vector generation for fault detection in MOS circuits.

Abstract-Level Approaches

While procedures have been developed to consider faults at the lowest component level, i.e., transistor level, other methods have been developed to view a circuit in a more abstract manner. Collectively, these techniques can be called abstract-level approaches. The benefit of these approaches is that by viewing a circuit in a more general way the computation inherent in automated diagnosis is reduced. Additionally, abstraction facilitates diagnosis of hierarchies in a circuit. In one view, a circuit may be one component in a system under diagnosis; in another, the circuit itself may be diagnosed. Abstract-level approaches include techniques based on graph theory, functional-level diagnostic techniques, and methods which use some form of circuit partitioning.

In methods which use graph theory for fault diagnosis, a combinational circuit is transformed into a graph representing the circuit. Principles of graph theory are applied to generate test vectors. Akers demonstrated the utility of graph theory for fault diagnosis in combinational circuits [Akers 74]. Wang developed a method based on graph theory which can be used both to generate single and multiple-fault detection test sets and to fabricate fault dictionaries [Wang 75].

Functional-level diagnostic techniques use a functional description of a circuit to facilitate diagnosis. Using the functional circuit description, a functional-level fault model is developed. This fault model "covers functional faults that alter the behavior of a module during one of its modes of operation" [Abadi 85:15]. Thatte and Abraham developed a functional-level diagnostic system for microprocessors [Agraw 88:21]. They used functional descriptions such as the system architecture or instruction set to develop fault models for the circuit under test. Types of faults that could be detected using their diagnostic system include instruction decoding errors, data storage and data transfer faults, and data manipulation errors [Agraw 88:21]. An advantage of functional-

level diagnosis is that the functional description of a system typically is smaller than the gate-level description of the same system [Agraw 88:21]. A functional description may be easier to manipulate by a diagnostic system than a gate-level representation; hence, diagnosis at the functional view of a circuit may be more tractable than gate-level diagnosis. On the other hand, fault resolution is not as good in a functional-level diagnostic system because the diagnostic system only may identify faulty components rather than faulty gates or nodes.

Diagnostic methods which use circuit partitioning incorporate the advantages of circuit abstraction. In Walczak and Sapiecha's method, a large combinational circuit is decomposed into smaller one-output subcircuits called units [Walcz 84]. Using another procedure which can generate test vectors for fault detection, test sets are developed to detect multiple stuck-at faults within each of the units. Test sets for individual units are combined to form what is called a macrotest set for the entire circuit. Using a path sensitization analysis, results from the application of the macrotest set to the circuit under test are used to deduce the state of units within the circuit [Walcz 84:138]. Units may be identified as either fault-free, faulty or undetermined. The output of this diagnostic system is the state of the units which constitute the circuit; hence, fault resolution is limited to the identification of faulty subcircuits. Because several units may be identified as faulty, Walczak and Sapiecha's method can be used to localize multiple faults within a circuit. Additionally, their method places no restrictions on the circuit to be diagnosed.

Abstract-level approaches differ from conventional techniques which are used to generate test sets for fault detection. Circuits may be dealt with in a hierarchical manner rather than at the gate-level. This can increase the efficiency of the diagnostic process as well as allow diagnosis of hierarchies in a circuit. Recent research has been conducted in the application of artificial intelligence techniques to fault diagnosis. An interesting aspect of these techniques is that they facilitate both hierarchical and gate-level diagnosis of circuits.

Artificial Intelligence Techniques

Diagnosis Based on Structure and Behavior. New approaches to fault diagnosis suggested in recent years use automated reasoning techniques associated with the domain of artificial intelligence to alleviate the problem of complexity. Davis describes a diagnostic system based on "a theory of reasoning that exploits knowledge of structure and behavior" in which the behavior of a system under test is used to determine faults in its structure [Davis 85:515]. Davis's approach departs from conventional fault diagnostic systems designed specifically to handle digital systems, because it is a general-purpose diagnostic method, i.e., it can diagnose systems other than digital systems. In an implementation of his method, however, Davis deals with diagnosis of digital circuits. Another important aspect of his approach is that it is designed specifically for system diagnosis, as opposed to other methods such as path sensitization which were designed only to generate test vectors for system validation [Davis 85:529].

Central to Davis's technique is the use of hierarchical design descriptions to model the system under test. Parallel sets of descriptions are used: one describing the behavior of the system under test, the other specifying the complete structure of the system. Descriptions are multi-layer; at higher levels a "black box" approach is taken, at lower levels a discrete component may be modeled [Davis 85:534]. Davis views faults as anything that causes a deviation between the actual behavior and the expected behavior of a system; faults are treated as "as modifications to the original design" of the system [Davis 85:519]. Hence, the goal of his method is to modify the design descriptions to produce a description that is consistent with the operation of the system under test. An advantage of this approach is that the diagnostic process yields a description of the potentially faulty system, even if the original schematic was wrong [Davis 85:537].

The method used to determine the final description of the circuit under test is called *constraint suspension*. In this method, the intended behavior of the system under test is a collection of constraints, or constraint network, in which each constraint corresponds to the expected behavior

of one of the system components. These constraints are derived from the initial component descriptions. During testing, constraints are removed from the collection of constraints until one retraction leaves the constraint network in a consistent state [Davis 85:518]. Hence, constraint suspension is the process of deducing which component is operating in a manner inconsistent with its original design descriptions. Inherent in this method is the necessity of a single-fault assumption, because only one constraint is removed from the constraint network at a time [Davis 85:576]. One explanation for the validity of this assumption is that if a device was operational at some prior time, it is likely to become faulty due to a single cause [Davis 85:576]. Additionally, even if many components appear to be faulty, the cause is likely to be traced to a single component due to the interaction of interconnected components [Davis 85:518].

Due to the manner in which faulty components are identified, no explicit fault model is required by Davis's method. This is important because the type of fault that can be diagnosed is unrestricted; the method can be used to diagnose classical and non-classical faults. Davis's method can be used to adaptively locate faults in combinational circuits of arbitrary structure. In addition, neither explicit enumeration of faults nor a transformation of the circuit representation is required. The most important weakness of the method is the constraint of a single fault assumption. Another limitation is that the method cannot be used for test generation for design validation.

The DART Program. Similar to Davis's method, the DART program developed by Genesereth performs diagnosis using hierarchical design descriptions. Multi-level descriptions of the structure and behavior of a system under diagnosis are used as the input to the DART program. After a set of observed malfunctions is input, DART generates test vectors, accepts the results of tests, and determines the faulty components. [Genes 84]

The component descriptions input to the DART program contain information about structure, e.g. devices and interconnections, or expected behavior, e.g. system equations [Genes 84:412]. All descriptions are propositions [Genes 84:414]. Using the propositions, DART employs direct proof

methods to compute a list of possible faulty components, generate tests vectors, derive information from the result of tests, and to pare the list of faulty components. Tests are conducted until DART fails to generate a test or eliminates all but one suspect [Genes 84:424].

Genesereth considers a fault to be a discrepancy between the actual system structure and its designed structure. An observed malfunction is placed in the form of a proposition; this proposition is inconsistent with the propositions that describe the correct operation of the device [Genes 84:419]. The goal of the DART program is to determine which proposition of the original design propositions should be negated; the negated proposition both makes the design propositions consistent with the malfunction proposition and identifies the faulty component. Only one of the original design propositions may be negated, because the DART program uses the single-fault assumption [Genes 84:418]. After the malfunction proposition is entered, a set of potential design propositions is generated corresponding to possible faulty components. Testing is used "to gather data to help confirm or disconfirm the propositions in the suspect set" [Genes 84:422]. Tests are generated in which the outcome is not known; such tests provide new information about the state of the system under diagnosis [Genes 84:423]. After a test, deduction is used to eliminate components from the list of possible faulty components. This process continues in an iterative fashion until a new test cannot be generated, or one suspect is left in the suspected fault list.

With respect to the diagnostic system evaluation criteria, the DART program has the same characteristics as Davis's diagnostic method. Like Davis's method, hierarchical descriptions are used to allow diagnosis at differing levels of abstraction in a design. Hence, the DART system overcomes problems associated with computational complexity for large designs [Genes 84:430]. Additionally, the use of propositions to model components allows the system to determine whether a component is good or bad regardless of the reason for the malfunction. Thus, the DART program is not constrained to a particular fault model.

Differing from Davis's method, DART demands a complete design description. Whereas Davis's method can be used to identify an incorrect description, the DART program will either run inefficiently or not at all if the description of the system to be diagnosed is incorrect [Genes 84:435]. Another disadvantage to Genesereth's approach is that many useless deductions are generated because direct proof techniques are used to arrive at conclusions. Genesereth does suggest several mechanisms for minimizing this inefficiency [Genes 84:425].

The General Diagnostic Engine. The General Diagnostic Engine (GDE) developed by deKleer and Williams differs from both DART and Davis's method by facilitating the diagnosis of multiple faults in a system under diagnosis. However, GDE is similar to these approaches; it begins with a model of the potentially-faulty system and uses the process of diagnosis to determine which components of the model are not consistent with the actual operation of the system [deKle 87:98]. In GDE a description of a malfunction is used to develop a set of candidates that may be the cause of the malfunction. Tests are used to reduce the set of candidate faults until the final set of suspected faulty components is determined [deKle 87:98].

The distinction between GDE and other methods is that GDE uses probabilistic information and information theory to guide the diagnostic process. Before diagnosis of a circuit commences, the probabilities of individual component failures must be determined [deKle 87:98]. Using information theory techniques, an evaluation function is created which uses the component failure probabilities to determine the cost of a specific test [deKle 87:114]. The evaluation function is used to determine the optimal test at each step in the test generation process. Hence, the faulty components are determined in a minimum number of measurements [deKle 87:113].

GDE makes an advance over other diagnostic systems by permitting the diagnosis of multiple faults. However, the requirement for probabilistic information regarding the component failure probabilities is restricting. In digital systems, such data may not be readily obtainable [Breue 76b:19] [Lala 85:22].

Summary

This chapter, a survey of existing fault-diagnostic methods, is intended to give an insight into the tasks involved in developing a new method. Existing diagnostic systems were discussed with respect to a set of evaluation criteria. Most systems have limitations which restrict their utility. For example, many methods can be used only for preset generation of fault detection test sets, i.e. they cannot locate faults. Adaptive systems typically locate faults, but are restricted for a variety of reasons. Such systems may be based on a single-fault assumption, may require a priori generation of fault detection test sets, or may require knowledge of probabilities of specific faults. No systems exist which can adaptively locate multiple faults without requiring a priori fault enumeration or knowledge of component fault probabilities.

In the next chapter, the evaluation criteria developed in this chapter are used to define the requirements of an "ideal" diagnostic system. Concepts taken from techniques discussed in this chapter are used to develop ideas regarding model-based diagnosis and the choice of model and reasoning method for diagnosis. The choice of programming language to be used in an implementation of the diagnostic procedure is also discussed.

III. Problem Analysis

Introduction

Before developing a new diagnostic system for combinational circuits, the attributes of an "ideal" diagnostic system should be defined. The evaluation criteria discussed in the previous chapter can be used as a basis for listing the features of an ideal system. A goal of developing a new method is to incorporate the attributes of an ideal diagnostic system. In practice, however, compromises are made because it is virtually impossible to realize all of the aspects of an ideal system.

An ideal diagnostic system would be an efficient, flexible tool that imposes no limits on the circuit designer. Most diagnostic systems can be used only to generate fault detection test sets, some produce fault dictionaries for preset fault location, while few perform adaptive fault location. A diagnostic procedure should be responsive enough to perform all of these tasks. Moreover, this type of flexibility should be extended to the type of circuit diagnosed by the system. Many systems place restrictions, such as irredundancy or constraints on topology, on the circuit to be diagnosed. Optimally, a circuit designer should not be restricted by the procedure that will be used to generate tests to validate his design. Additionally, regardless of the task it is performing and independent of the structure of the circuit, the system should generate a minimal set of tests.

While maintaining flexibility, an ideal system should diagnose multiple faults without requiring a priori fault enumeration. A single-fault assumption may not be valid as circuits gain in complexity; yet, there are too many multiple faults to enumerate in a large circuit. Furthermore the multiple-fault case should be inherent throughout the test generation process. Tests should be generated to detect or locate multiple faults; this is in distinction to systems which generate tests for single faults, perform some type of masking analysis, and then generate tests to cover selected multiple faults.

Increasing circuit complexities have been the result of new technologies. Implicit to these technologies are faults that differ from the classical fault model. These new types of faults may not be diagnosed by tests generated to detect or locate classical faults [Baner 84, Bate 88, Burge 88]. Thus, all types of faults, classical and nonclassical, should be diagnosed by the diagnostic system. Furthermore, an ideal system would be able to diagnose nonclassical faults without any transformation of the circuit representation. This would eliminate unnecessary processing as well as prevent inconsistencies that may result from the translation.

Finally, an ideal diagnostic procedure would be able to diagnose faults at differing levels of a circuit's topology. Thus, a system should be able to detect and locate a fault at some abstract, functional level as well as at the lowest component level.

A summary of the attributes of an ideal diagnostic system is:

1. Detects and Locates Multiple Faults.
2. Operates in Preset Mode or Adaptively.
3. Generates Minimal Test Sets.
4. Diagnoses Circuits of any Structure.
5. Requires No Fault Enumeration.
6. Performs No Masking Analysis.
7. Does Not Transform Circuit Representation.
8. Diagnoses Classical and Nonclassical Faults.
9. Diagnoses Faults at all Levels of a Circuit Hierarchy.

Model-Based Diagnosis

Most diagnostic systems generate test sets which will detect faults but not locate them. Davis states that these systems are limited because they are based on "a theory of *test generation*, not a theory of *diagnosis*" [Davis 85:528]. A theory of test generation is one in which a fault is specified and then tests are generated to detect the fault [Davis 85:528]. All test generation systems which can be used only in a preset manner to generate test vectors are based on a theory of test generation. A system which can adaptively generate test vectors must take a different approach.

An adaptive diagnostic system must use knowledge gained from previous tests to narrow down the number of tests that are required to detect or locate faults. Such a system is based on a theory of diagnosis, "since it allows systematic isolation of possibly faulty devices, and does so without having to precompute fault dictionaries, diagnosis trees, or the like" [Davis 85:534].

Essential to developing an adaptive system is the development of a model of the circuit to be diagnosed. Typically, this model is some representation of the structure of the circuit. Given such a model, the purpose of adaptive diagnosis is to use it in conjunction with the actual circuit behavior to determine faults in the circuit as well as a representation of the actual circuit function [Genes 84:419]. Thus, adaptive testing deduces information about the state of the circuit model. Once the testing process is concluded, the model yields information about faults and actual circuit function. In some approaches, the diagnostic process attempts to find inconsistencies in the model [Genes 84]. An inconsistency may represent a faulty component. Others use a method called *constraint propagation* to derive information about the model [Davis 85]. In systems which use constraint propagation, the model of the circuit has variables which are in an unknown state at the outset of testing. Knowledge gained from each test, i.e., the circuit's response to the test, limits the possible values that the variables may take. The values that the variables may take are constrained by the circuit's behavior; the variables cannot assume values that are inconsistent with the behavior of the circuit. After a sufficient number of tests, a given variable's value may be derived. Once this occurs, variables dependent on this variable are in turn limited. This is called propagation of the constraint [Tanim 87:124]. At the conclusion of testing, constraint propagation yields the value of all of the variables, if determinable. In diagnosis, these values indicate the possible presence of faults.

The process of reasoning is used to generate tests which can derive information about a model. In a diagnostic system, the type of reasoning used and the model are normally interdependent. A model that includes variables, the values of which must be determined, will use a method which

can find these values. Models which use propositions or predicates to describe system components use theorem proving techniques to determine the consistency of the propositions. Hence, for every type of model which is used to represent a circuit there is a corresponding form of reasoning that must be used to gain knowledge about the state of the model. Additionally, regardless of the type of reasoning used, the outcome of every test that is generated should not be predictable. Only under this circumstance can new information be derived from a test [Genes 84:423]. A test which meets this criterion is called an *effective test*.

The Choice of Model and Reasoning Method

The first steps in designing a diagnostic system that can adaptively diagnose faults are to develop a circuit model and choose a corresponding reasoning method. The reasoning method is used to generate effective tests and to employ the results of tests to derive information about the model, resulting in knowledge about the state of faults in the network as well as the actual circuit function. Likewise, when choosing the model and the reasoning method, the attributes of an ideal diagnostic system should be considered.

Several different approaches are amenable to adaptive fault diagnosis. These include selected Boolean algebraic methods, line-deduction procedures, and techniques grounded in artificial intelligence. Adaptive diagnostic systems have been developed using all of these methods. However, in virtually all cases, restrictions that we would like to overcome are imposed on these systems. These restrictions include the single-fault assumption, requirements to generate a fault detection test set prior to an adaptive experiment, and the necessity of determining the probabilities of possible faults. The approach to be taken in this project is to extend an existing algebraic technique to perform adaptive fault diagnosis without restrictions.

Only one algebraic approach, Breuer, Chang, and Su's on-line method, can be used for fault location. The circuit model that they use to generate tests is the cause-effect equation. In their

system, the reasoning techniques used are the solving of Boolean equations and the comparison of functions using the XOR operator [Breue 76a]. Their procedure is not adaptive because of the necessity to generate a fault-detection test set prior to the execution of an input-output experiment. The results of the experiment are used to generate another test for the circuit. The outcome of this test is used to gain more information about the circuit. This process iterates until the state of the circuit is determined. Breuer, Chang, and Su's method is demonstrated in Example 3.1.

Example 3.1:

To diagnose the circuit in Figure 3.1, Breuer, Chang, and Su first need to develop the cause-effect equation which represents the circuit. The checkpoints in the circuit are nodes a , b , and c . For each checkpoint, there are three related checkpoint variables. For example, node a has variables a_0 , a_1 , and a_n representing the following conditions:

- $a_n = 1$ if line a is normal, 0 otherwise,
- $a_0 = 1$ if line a is stuck-at-0, 0 otherwise,
- $a_1 = 1$ if line a is stuck-at-1, 0 otherwise. [Poage 63:487]

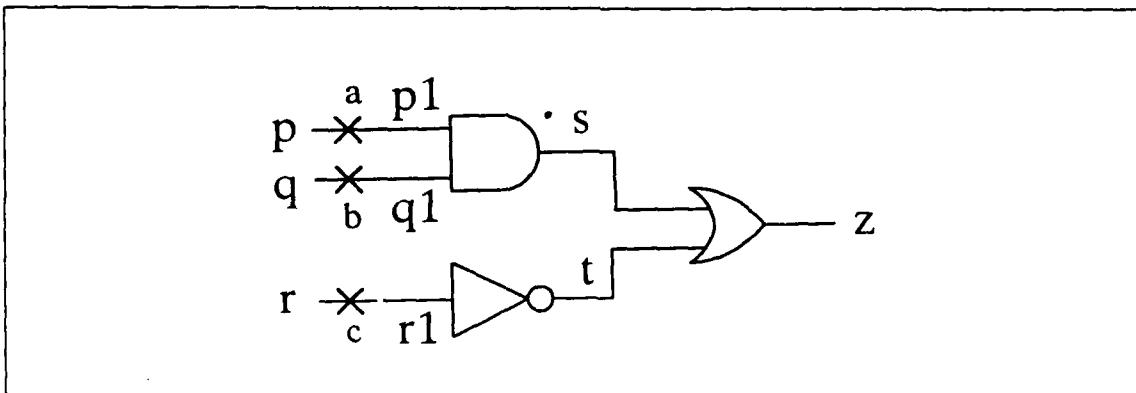


Figure 3.1. Circuit for Example 3.1

These variables are related by the following equations:

$$a_0 a_1 = a_1 a_n = a_0 a_n = 0 \quad (3.1)$$

$$a_0 + a_1 + a_n = 1. \quad (3.2)$$

[Bosse 71:1253]

Each line in the circuit which is not a checkpoint is represented by the following equations:

$$a_{out} = a_{in} \quad (3.3)$$

$$a'_{out} = a'_{in}. \quad (3.4)$$

[Bosse 71:1253]

Lines which are checkpoints are represented as follows:

$$a_{out} = a_n \cdot a_{in} + a_1 \quad (3.5)$$

$$a'_{out} = a_n \cdot a'_{in} + a_0. \quad (3.6)$$

[Bosse 71:1253]

Using these equations, the cause-effect equation for the circuit in Figure 3.1 is derived in the following manner:

$$\begin{aligned} a_{in} &= p \\ b_{in} &= q \\ c_{in} &= r \\ a_{out} &= a_n \cdot a_{in} + a_1 \\ b_{out} &= b_n \cdot b_{in} + b_1 \\ c'_{out} &= c_n \cdot c'_{in} + c_0 \\ s_{in} &= a_{out} \cdot b_{out} \\ t_{in} &= c'_{out} \\ s_{out} &= s_{in} \\ t_{out} &= t_{in} \\ z_{in} &= s_{out} + t_{out} \\ z_{out} &= z_{in}. \end{aligned} \quad (3.7)$$

Combining these equations yields z_{out} in terms of the input signals and checkpoint variables—the cause-effect equation. The cause-effect equation for Figure 3.1 is:

$$z_{out} = a_n b_n p q + a_n b_1 p + a_1 b_n q + a_1 b_1 + c_n r' + c_0. \quad (3.8)$$

Bossen and Hong's procedure, or any which generates test sets for multiple faults, is then used to generate a complete fault-detection test set, $D = \{(0, 1, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)\}$. This test set is applied to a potentially faulty realization of the circuit in Figure 3.1. If node a were stuck-at-1, then the results of the input-output experiment on the circuit would be as shown in Table 3.1.

p	q	r	$f(p, q, r)$
0	1	0	1
0	1	1	1
1	0	1	0
1	1	1	1

Table 3.1. Results of Input-Output Experiment

These results are substituted into the cause-effect equation to yield a system of equations with respect to the checkpoint variables—one equation for each test. For the given results, these equations are:

$$\begin{aligned} 1 &= a_1 b_n + a_1 b_1 + c_n + c_0 \\ 1 &= a_1 b_n + a_1 b_1 + c_0 \\ 0 &= a_n b_1 + a_1 b_1 + c_0 \\ 1 &= a_n b_n + a_n b_1 + a_1 b_n + a_1 b_1 + c_0. \end{aligned} \quad (3.9)$$

Equation (3.9), and checkpoint equations of the form given by (3.1) and (3.2), are combined using algebraic techniques to form a single Boolean equation:

$$a'_0 a_1 a'_n b'_0 b'_1 b_n c'_0 c_1 c'_n + a'_0 a_1 a'_n b'_0 b'_1 b_n c'_0 c'_1 c_n = 1. \quad (3.10)$$

Each uncomplemented literal in equation (3.10) is equal to 1. Omitting the complemented literals, (3.10) can be rewritten in the following way:

$$\begin{aligned} a_1 b_n c_1 &= 1 \\ a_1 b_n c_n &= 1 \end{aligned} \quad (3.11)$$

These results are then substituted into the cause-effect equation to obtain fault functions. Fault functions are the functions that the potentially faulty circuit may be implementing. The fault functions obtained in this case are:

$$\begin{aligned} F_1 \Big|_{a_1 b_n c_1 = 1} &= q \\ F_2 \Big|_{a_1 b_n c_n = 1} &= q + r' \end{aligned} \quad (3.12)$$

To differentiate between the functions, a test must be constructed that will divide the functions into two categories. This is done by taking the exclusive-or, XOR, of the functions to yield a new function which embodies the differences between the initial functions. A minterm of this new function is used as a test for the original functions as well as the circuit under test [Breue 76a:46].

In this instance, this test is generated as follows:

$$\begin{aligned} q \oplus (q + r') &= q'(q + r') + q(q + r')' \\ &= q'q + q'r' + q(q'r) \\ &= q'r \end{aligned} \quad (3.13)$$

Since we desire a minterm of this new function, we can choose either $p'q'r'$ or pqr' . We will choose $p'q'r'$. Thus, the new test generated is $(0, 0, 0)$. This test is input to the fault functions, F_1 and F_2 , as well as the circuit under test. The results of this test are given in Table 3.2.

Item Tested	Result
F_1	0
F_2	1
Circuit	1

Table 3.2. Results of New Test

Since F_2 and the circuit under test acted similarly with the new test that was generated, we discard function F_1 and conclude that F_2 is the function that the circuit under test is implementing. If there existed more than two functions, then two functions would be arbitrarily chosen to generate the new test with the XOR operator. The test would be used on all of the functions as well as the circuit under test. The functions which had the same result for the test as the circuit would be kept, the remainder discarded. The process would be iterated until a single fault function was found to act the same as the circuit under test [Breue 76a:49]. This fault function provides an equation which describes the operation of the faulty circuit as well as the fault conditions in the circuit. Recall that in equation (3.12), the fault functions were formed by substituting fault conditions into the cause-effect equation. Hence, the set of fault conditions used to form the final fault function are the fault conditions which may exist in the circuit. \square

Breuer, Chang, and Su's technique provides the basis for a truly adaptive procedure for multiple-fault location. The cause-effect equation provides the structural model of the circuit that is necessary to perform diagnosis down to gate level. However, a different approach to reasoning with respect to Boolean equations must be taken. In such an approach, mechanisms must be developed to generate tests based on the circuit structure, the checkpoint variables, as well as the results of previous tests. Boolean reasoning, reduction, and elimination provide tools that can be used for such a purpose.¹ The system developed in this project will use these techniques.

A diagnostic system based on the cause-effect equation model and Boolean reasoning can meet most, but not all of the criteria that define an ideal diagnostic system. Due to the choice of

¹See Appendix B.

the cause-effect equation as the circuit model, two limitations are immediately imposed. Because the cause-effect equation was developed to generate tests to detect stuck-at faults, only this type of fault may be located by the proposed system. Furthermore, the cause-effect equation is generated using a gate-level circuit description; hence, it can be used to localize faults only at gate-level. Since the best known techniques only generate tests to detect single stuck-at faults at the gate level, we will accept these constraints in this case. Overcoming these limitations is a challenge for further research. The algorithm used in the system as well as the system implementation will establish how the diagnostic system will meet the remaining criteria of an ideal diagnostic system. How these criteria are met by the system developed in this project will be discussed in later chapters.

The Implementation Language

The choice of implementation language for the system is dependent on the choice of model and reasoning method. Since an algebraic model and Boolean reasoning techniques are to be used, a language that can manipulate Boolean equations is most desirable. Boolean equations are different from conventional equations because they are processed symbolically rather than numerically. Conventional languages such as FORTRAN and C handle numerical calculations very efficiently; implementing symbolic processes in these languages is a laborious task [Eisen 88:2]. Symbolic languages enable rapid program development; this allows concentration on the problem and not the programming. Since the objective of this project is to obtain a working diagnostic system, a symbolic language is best suited to the task.

Considerations regarding the choice of a particular symbolic language include portability, standardization, ease of programming, and efficiency. To insure portability, implementations of the language must be available in a variety of environments from PC-based systems to mainframes [Fause 86:17]. Programs which are written to execute in one environment must work on language implementations in a totally different environment. This insures that potential users of the diag-

nostic system can apply it on the machines available to them. Thus, languages implemented on special-purpose hardware, e.g., LISP machines, are undesirable for implementation of the diagnostic system.

Associated with portability is standardization. One of the keys to portability of programs written in a particular language is the standardization of the language. A clear, concise standard should exist for an implementation language. Programs written to conform to such a standard are likely to run on implementations of the language in varying environments. Variations from the standard can be documented; when a program is to be ported to a differing environment, minor changes can be made to insure that it will operate in the new setting. [Fause 86, Rees 86]

Ease of programming allows a programmer to easily learn and program with a given language. This includes writing original code in the language as well as modifying existing code to perform a given task. A language is efficient if the code that the programmer creates executes quickly while minimizing memory utilization. The language chosen to implement the diagnostic system must facilitate ease of programming and produce efficient code. This will allow concentration on the problem to be implemented rather than the programming effort.

Symbolic languages include LISP and PROLOG. Common Lisp is the most widely used form of LISP. Standards exist for Common Lisp; however, it does not execute very efficiently on general-purpose computer systems. Common Lisp interpreters are typically found on hardware systems developed as dedicated LISP machines [Fause 86:18]. Furthermore, Common Lisp is very difficult to learn, and debugging programs written in Common Lisp is an arduous task.

In general, PROLOG meets the criteria considered for the choice of implementation language. Standardized versions are available on a variety of computer systems. Hence, PROLOG code can be written so that it is portable between different environments. PROLOG is also an easy language to learn. However, procedures written in PROLOG are inefficient, particular with respect to memory

utilization. Even a moderately-sized diagnostic system can exhaust available memory in short order [Brown 88b].

Scheme is a modern dialect of LISP that does not have the limitations of Common Lisp. Versions of Scheme are available on a wide range of systems including personal computers, mini-computers, and mainframes. Current variations all conform to a standard defining the Scheme language—the Revised³ Report on the Algorithmic Language Scheme [Rees 86]. Moreover, Scheme was designed to be easy to learn and use [Eisen 88:2]. At the Massachusetts Institute of Technology, Scheme is used in the introductory computer science course taken by all electrical engineering and computer science students [Abels 85]. Additionally, Scheme is generally more efficient than Common Lisp. A prime example of this is that Scheme implementations “are required to be properly tail-recursive” in order to be called “Scheme” [Rees 86:3]. Eisenberg defines a tail-recursive procedure as:

...one in which the value of the recursive call provides the complete result of the original call. In other words, once the recursive call is evaluated, there is no additional work to do to find the result of the original call. [Eisen 88:51]

The implication of tail-recursion is that iterative processes may be implemented recursively; yet, the computation is performed in constant space [Rees 86:3]. Essentially, when a recursive procedure calls itself by a tail-recursive procedure call, a new version of the procedure supersedes the previous call in memory, thus limiting memory usage. Hence, some algorithms that take prohibitive amounts of memory in some languages occupy acceptable space when implemented in Scheme.

In his thesis, Faucett compared the efficiency of implementing Boolean operations with different data structures in the Scheme language [Fause 86]. Brown has determined how to implement specific Boolean operations in the most efficient manner in Scheme [Brown 88b]. In both cases, the Scheme language has been shown to be an extremely useful tool for implementing Boolean problem-solving techniques.

Conclusion

A symbolic language should be used to implement a diagnostic system using a model based on Boolean equations. Given the criteria of portability, standardization, ease of programming, and efficiency, Scheme is the most suitable language. Additionally, Scheme has proven to be useful in implementing Boolean operations. Thus, Scheme has been used to implement the diagnostic system.

A model of the circuit as well as a reasoning method must be developed to realize a diagnostic system. Breuer, Chang, and Su's on-line method, featuring the cause-effect equation, provides the basis for adaptive diagnosis. However, a different reasoning method must be used. Boolean reasoning provides all of the mechanisms for an adaptive system based on Boolean equations. While the cause-effect equation imposes the limitation that only classical faults may be diagnosed at the gate level, the algorithm used in the system will establish how the diagnostic system will meet the remaining criteria of an ideal diagnostic system. The next chapter provides the development of this algorithm.

IV. Mathematical Development of the Diagnostic System

In this chapter, the mathematical basis for the diagnostic system is developed. However, to limit the scope of this project, this derivation is limited to diagnosis of single-output combinational circuits. Extension of the algorithm for multiple-output circuits is a subject for future work.

Revision of the Checkpoint Model for Stuck-at Faults

In Breuer, Chang, and Su's method for fault location, a checkpoint node a has three associated checkpoint variables:

- $a_n = 1$ if line a is normal, 0 otherwise,
- $a_0 = 1$ if line a is stuck-at-0, 0 otherwise,
- $a_1 = 1$ if line a is stuck-at-1, 0 otherwise. [Poage 63:487]

These checkpoint variables are related by the following equations:

$$a_0 a_1 = a_1 a_n = a_0 a_n = 0 \quad (4.1)$$

$$a_0 + a_1 + a_n = 1. \quad (4.2)$$

[Bosse 71:1253]

When constructing the cause-effect equation, lines which are checkpoints are represented as follows:

$$a_{out} = a_n \cdot a_{in} + a_1 \quad (4.3)$$

$$a'_{out} = a_n \cdot a'_{in} + a_0, \quad (4.4)$$

where a_{in} is the input to a checkpoint, and a_{out} is the output of a checkpoint. [Bosse 71:1253]

There exist weaknesses in this approach. One problem is that each checkpoint is represented by three checkpoint variables. A reduction in the number of variables which must be dealt with increases the efficiency of the algebraic manipulation that must be performed by the diagnostic

system. A second problem is that three equations must be considered when determining the state of the checkpoint variables. Either equation (4.3) or equation (4.4) is used in the formation of the cause-effect equation; equations (4.1) and (4.2) are used in the formation of a single Boolean equation representing the possible faults conditions in the circuit.¹ A reduction of the number of equations to be considered may also increase the efficiency of the system. To overcome these weaknesses, an alternative representation was developed by Brown to represent each checkpoint by two variables and two equations [Brown 88b].

In the revised mathematical model for a checkpoint node a , there are two checkpoint variables a_0 and a_1 . The output of a checkpoint node is dependent on the states of the checkpoint variables as well as the input to the checkpoint node. Hence, we can view a checkpoint node as a new form of logic gate depicted in Figure 4.1.

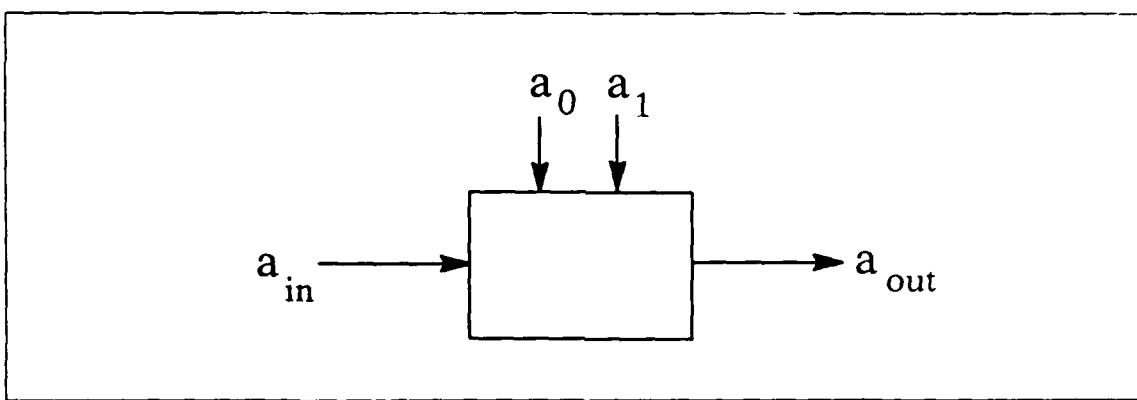


Figure 4.1. Checkpoint Logic Gate

The truth table given in Table 4.1 defines the function of the checkpoint logic gate for the conditions of the checkpoint variables on node a . Note that both a_0 and a_1 cannot take the value of 1 simultaneously because a line cannot be stuck-at-0 and stuck-at-1 concurrently.

The function, a_{out} , of the checkpoint logic gate is given by the Karnaugh map in Figure 4.2.

¹See equation (3.10) in Example 3.1.

$a_0 \ a_1$	Output	Node Conditions
0 0	a_{in}	Node is Normal
0 1	1	Node is Stuck-at-1
1 0	0	Node is Stuck-at-0
1 1	—	Cannot Occur

Table 4.1. Truth Table for the Checkpoint Logic Gate.

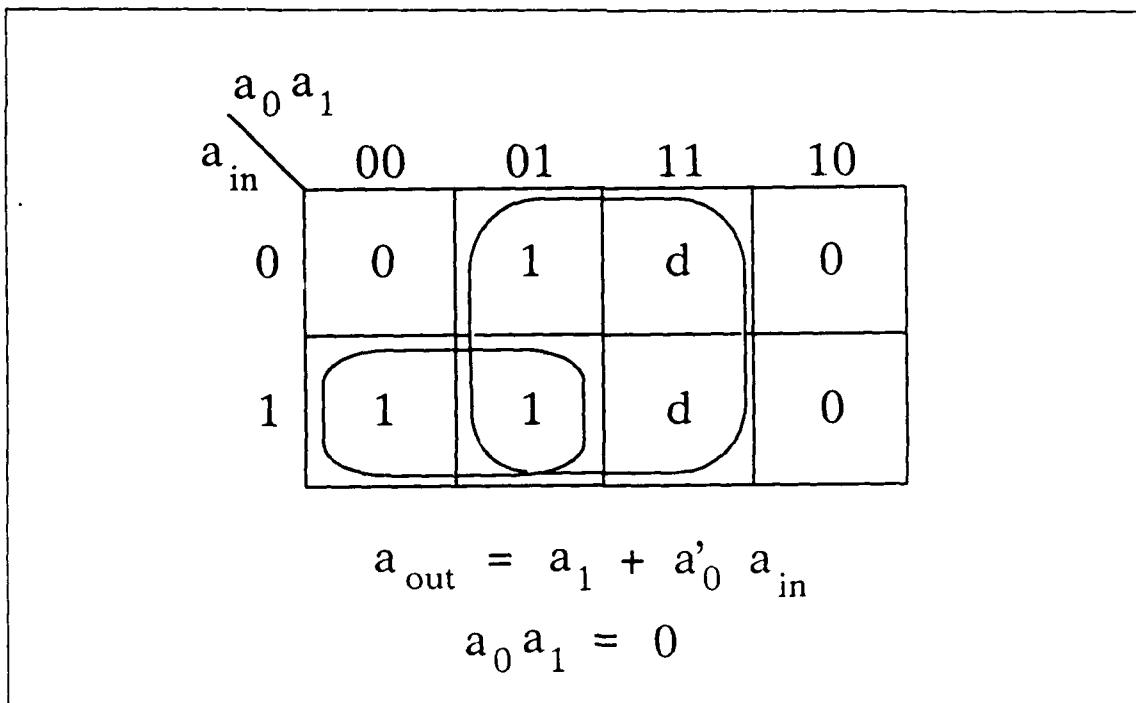


Figure 4.2. Karnaugh Map for the Checkpoint Logic Gate

From the Karnaugh map, the following equation is derived to represent the checkpoint logic gate:

$$a_{out} = a_1 + a'_0 a_{in}. \quad (4.5)$$

An equation representing a constraint on equation (4.5) is

$$a_0 a_1 = 0. \quad (4.6)$$

Another way of depicting a checkpoint logic gate is as a combination of conventional AND and OR gates. This view is given in Figure 4.3. The constraint given by equation (4.6) holds in this case also.

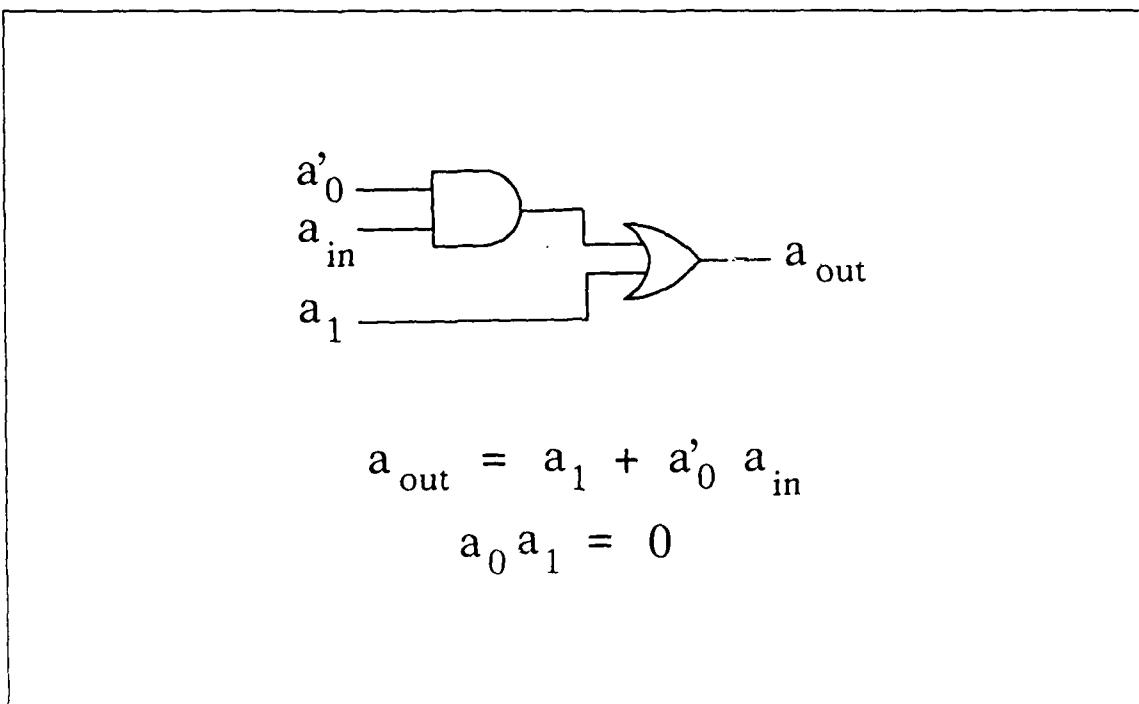


Figure 4.3. Alternate View of a Checkpoint Logic Gate

Derivation of a Characteristic Equation

Using the checkpoint logic gate representation, a single Boolean equation can be developed which represents a circuit to be diagnosed. Suppose a circuit is represented by Figure 4.4. The locations of checkpoints are as defined by Bossen and Hong. In Figure 4.4, checkpoint nodes are denoted by y_1, y_2, y_3 , and y_4 .

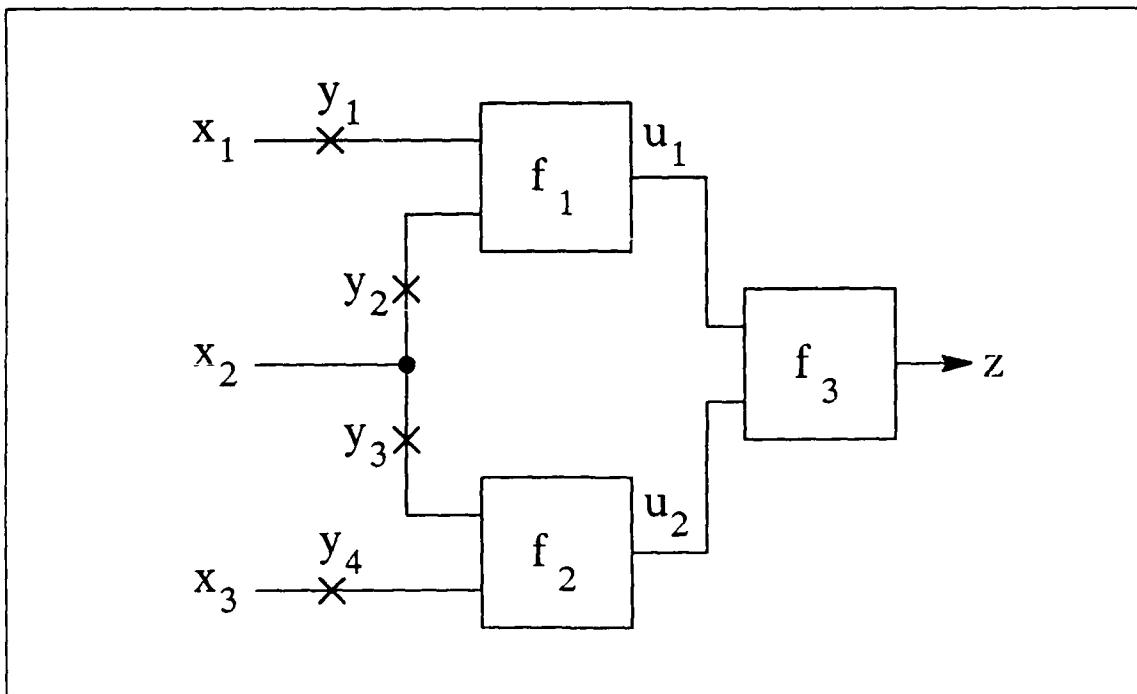


Figure 4.4. Circuit Used in Equation Derivation

In place of the checkpoint nodes, checkpoint logic gates are inserted as shown in Figure 4.5.

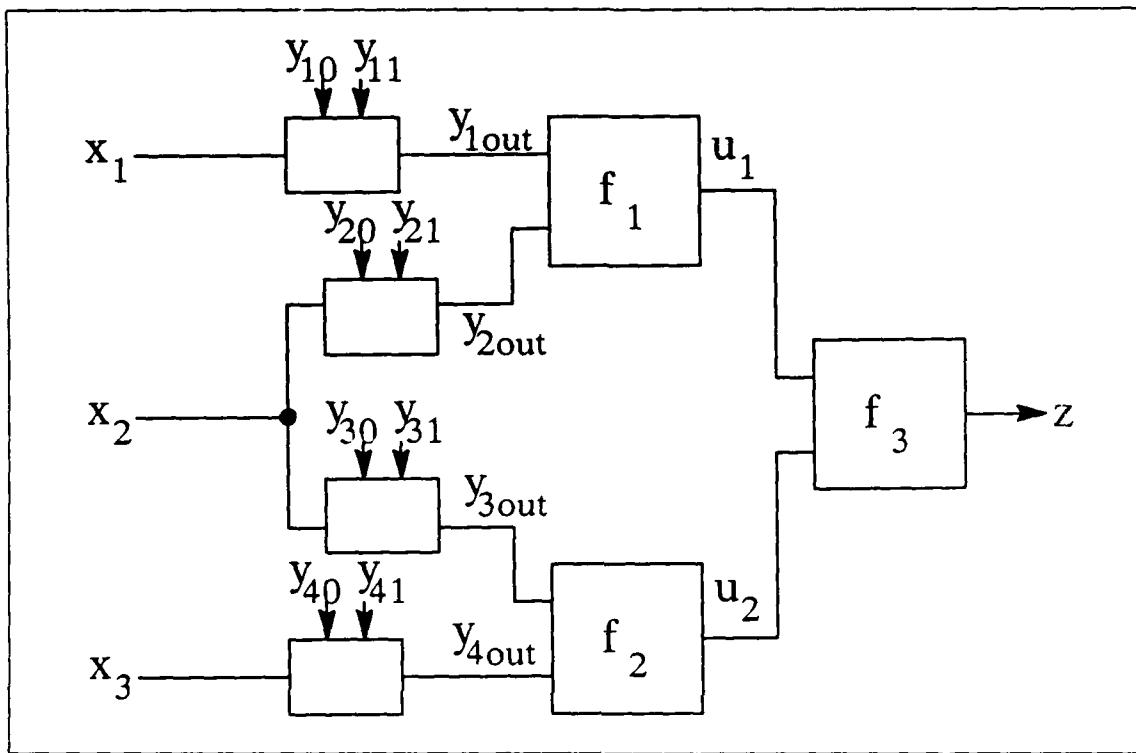


Figure 4.5. Circuit with Checkpoint Logic Gates Inserted

The circuit in Figure 4.5 is represented by Boolean equations as follows:

$$\begin{aligned}
 z &= f_3(u_1, u_2) \\
 u_1 &= f_1(y_{1out}, y_{2out}) \\
 u_2 &= f_2(y_{3out}, y_{4out}) \\
 y_{1out} &= y_{11} + y'_{10}x_1 \\
 y_{2out} &= y_{21} + y'_{20}x_2 \\
 y_{3out} &= y_{31} + y'_{30}x_2 \\
 y_{4out} &= y_{41} + y'_{40}x_3
 \end{aligned} \tag{4.7}$$

where

- y_{10} and y_{11} are the checkpoint variables associated with checkpoint node y_1 ,
- y_{20} and y_{21} are the checkpoint variables associated with checkpoint node y_2 ,
- y_{30} and y_{31} are the checkpoint variables associated with checkpoint node y_3 , and
- y_{40} and y_{41} are the checkpoint variables associated with checkpoint node y_4 .

Using Boolean reduction, this system of equations can be reduced to a single equation. The derivation is shown below.

$$\begin{aligned}
 z \oplus f_3(u_1, u_2) &= 0 \\
 u_1 \oplus f_1(y_{1out}, y_{2out}) &= 0 \\
 u_2 \oplus f_2(y_{3out}, y_{4out}) &= 0 \\
 y_{1out} \oplus y_{11} + y'_{10}x_1 &= 0 \\
 y_{2out} \oplus y_{21} + y'_{20}x_2 &= 0 \\
 y_{3out} \oplus y_{31} + y'_{30}x_2 &= 0 \\
 y_{4out} \oplus y_{41} + y'_{40}x_3 &= 0
 \end{aligned} \tag{4.8}$$

By equations (B.48) and (B.49) this system is converted to $f = 0$ form.

$$\begin{aligned}
 &(z \oplus f_3(u_1, u_2)) + \\
 &(u_1 \oplus f_1(y_{1out}, y_{2out})) + \\
 &(u_2 \oplus f_2(y_{3out}, y_{4out})) + \\
 &(y_{1out} \oplus y_{11} + y'_{10}x_1) + \\
 &(y_{2out} \oplus y_{21} + y'_{20}x_2) + \\
 &(y_{3out} \oplus y_{31} + y'_{30}x_2) + \\
 &(y_{4out} \oplus y_{41} + y'_{40}x_3) = 0
 \end{aligned} \tag{4.9}$$

The mathematical model used for the checkpoint logic gate dictates constraints on the checkpoint variables. In this case, these constraints are given by

$$y_{11}y_{10} = y_{21}y_{20} = y_{31}y_{30} = y_{41}y_{40} = 0. \tag{4.10}$$

These constraints are converted to a single equality using the property demonstrated by equation (B.32), i.e.,

$$y_{11}y_{10} + y_{21}y_{20} + y_{31}y_{30} + y_{41}y_{40} = 0. \quad (4.11)$$

By the same property, this equation can be combined with equation (4.9) to form an equation of the form $f = 0$ representing the circuit as well as constraints on the checkpoint variables:

$$\begin{aligned} & (z \oplus f_3(u_1, u_2)) + \\ & (u_1 \oplus f_1(y_{1out}, y_{2out})) + \\ & (u_2 \oplus f_2(y_{3out}, y_{4out})) + \\ & (y_{1out} \oplus y_{11} + y'_{10}x_1) + \\ & (y_{2out} \oplus y_{21} + y'_{20}x_2) + \\ & (y_{3out} \oplus y_{31} + y'_{30}x_2) + \\ & (y_{4out} \oplus y_{41} + y'_{40}x_3) + \\ & (y_{11}y_{10} + y_{21}y_{20} + y_{31}y_{30} + y_{41}y_{40}) = 0. \end{aligned} \quad (4.12)$$

In diagnosis, we are interested only in the relationships between circuit inputs, \underline{x} , the circuit output, z , and the checkpoint variables, \underline{y} . The output of the circuit is a function of the input vector applied to the circuit as well as on the state of the checkpoint variables. Hence, all variables other than inputs, the output, and the checkpoint variables in equation (4.12) may be eliminated. In this case variables which can be eliminated are $u_1, u_2, y_{1out}, y_{2out}, y_{3out}$, and y_{4out} .

Because equation (4.12) is in $f = 0$ form, conjunctive elimination is used for eliminating the unnecessary variables. The conjunctive eliminant of the function f , represented by the formula on the left-hand side of the equality in equation (4.12), with respect to the unnecessary variables is given by $ECON(f, \{u_1, u_2, y_{1out}, y_{2out}, y_{3out}, y_{4out}\})$. This eliminant is a function of the circuit inputs, \underline{x} , the circuit output, z , and the checkpoint variables, \underline{y} . By equation (B.59), a simple way to construct $ECON(f, S)$, is to determine the Blake canonical form of f and then to remove the terms which include literals of the variables that are to be eliminated, i.e., the variables belonging

to set S . Thus, the function f is expressed in Blake canonical form; then all terms which have literals involving $u_1, u_2, y_{1out}, y_{2out}, y_{3out}$, and y_{4out} are removed.

We will define the eliminant $ECON(f, \{u_1, u_2, y_{1out}, y_{2out}, y_{3out}, y_{4out}\})$ as Φ . By (B.62), the new function is set equal to 0 to form the equation

$$\Phi(\underline{x}, \underline{y}, z) = 0 \quad (4.13)$$

where

- \underline{x} are all input variables,
- \underline{y} are all checkpoint variables, and
- z is the output variable.

Implicit in equation (4.13) is all information with regard to the function implemented by and the fault conditions of the circuit of Figure 4.4. Since this is a complete representation of the condition of the circuit, we will call $\Phi(\underline{x}, \underline{y}, z)$ the *characteristic function*. Equation (4.13) is called the *characteristic equation* of the circuit. The term "characteristic" was used in a similar manner by Cerny when he derived the Boolean functions of nodes in a circuit with respect to the circuit inputs [Cerny 76]. The initial characteristic function will be denoted $\Phi_0(\underline{x}, \underline{y}, z)$. After each test in the process of diagnosis, the characteristic function will be updated to form a revised characteristic function. The characteristic function after i tests is given by $\Phi_i(\underline{x}, \underline{y}, z)$. The entire process of deriving an initial characteristic equation for a circuit is demonstrated in Example 4.1.

Example 4.1:

Given the circuit shown in Figure 3.1, the circuit can be redrawn with checkpoint logic gates. This circuit is given in Figure 4.6.

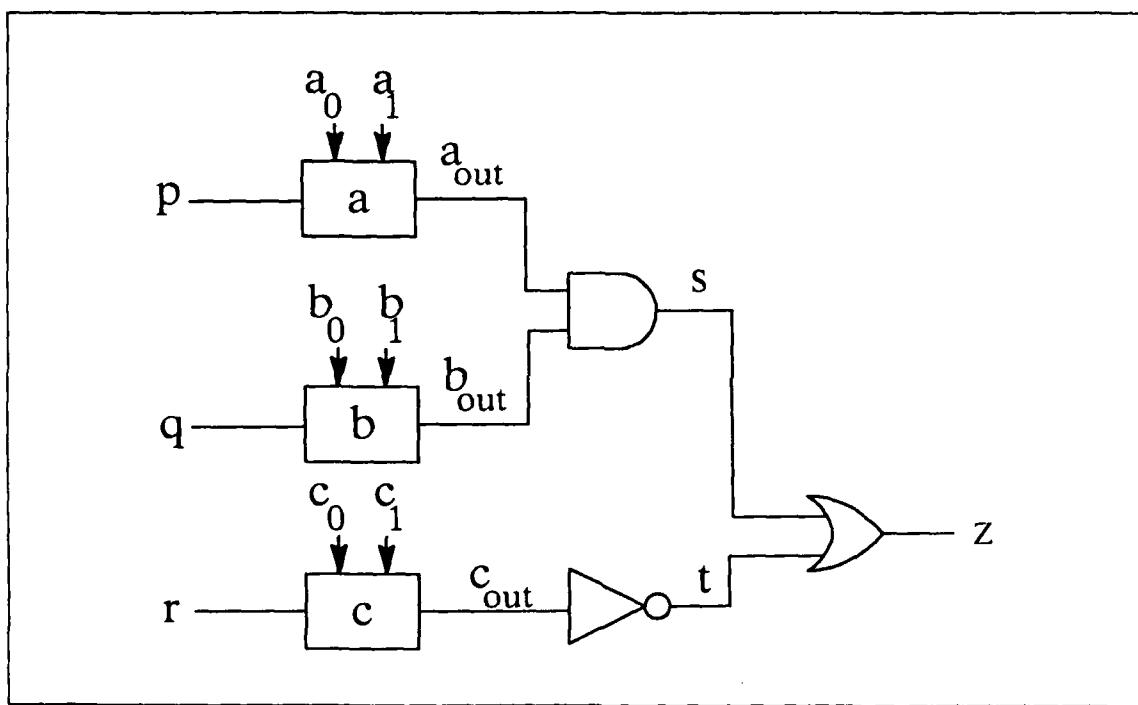


Figure 4.6. Circuit for Example 4.1

This circuit is represented by the equations

$$\begin{aligned}
 z &= s + t \\
 s &= a_{out} b_{out} \\
 t &= c'_{out} \\
 a_{out} &= a_1 + a'_0 p \\
 b_{out} &= b_1 + b'_0 q \\
 c_{out} &= c_1 + c'_0 r.
 \end{aligned} \tag{4.14}$$

These equations are rewritten as

$$\begin{aligned}
 (z \oplus (s + t)) &= 0 \\
 (s \oplus (a_{out} b_{out})) &= 0 \\
 (t \oplus (c'_{out})) &= 0 \\
 (a_{out} \oplus (a_1 + a'_0 p)) &= 0 \\
 (b_{out} \oplus (b_1 + b'_0 q)) &= 0 \\
 (c_{out} \oplus (c_1 + c'_0 r)) &= 0.
 \end{aligned} \tag{4.15}$$

This system of equations can be expanded and combined to form a single equation

$$\begin{aligned}
 a_{out} a'_1 p' + a_{out} a'_1 a_0 + a'_{out} a'_0 p + a'_{out} a_1 + \\
 b_{out} b'_1 q' + b_{out} b'_1 b_0 + b'_{out} b'_0 q + b'_{out} b_1 + \\
 c_{out} c'_1 r' + c_{out} c'_1 c_0 + c'_{out} c'_0 r + c'_{out} c_1 + \\
 b'_{out} s + a'_{out} s + a_{out} b_{out} s' + c_{out} t + c'_{out} t' + \\
 s z' + t z' + s' t' z &= 0.
 \end{aligned} \tag{4.16}$$

The constraints on the checkpoint variables are given by

$$a_1 a_0 + b_1 b_0 + c_1 c_0 = 0. \tag{4.17}$$

Equations (4.17) and (4.17) are combined to form a new equation

$$\begin{aligned}
 & a_{out}a'_1p' + a_{out}a'_1a_0 + a'_{out}a'_0p + a'_{out}a_1 + \\
 & b_{out}b'_1q' + b_{out}b'_1b_0 + b'_{out}b'_0q + b'_{out}b_1 + \\
 & c_{out}c'_1r' + c_{out}c'_1c_0 + c'_{out}c'_0r + c'_{out}c_1 + \\
 & b'_{out}s + a'_{out}s + a_{out}b_{out}s' + c_{out}t + c'_{out}t' + \\
 & sz' + tz' + s't'z + \\
 & a_1a_0 + b_1b_0 + c_1c_0 = 0.
 \end{aligned} \tag{4.18}$$

Expressing the left-hand side of (4.19) in Blake canonical form produces the equation

$$\begin{aligned}
 & a_{out}a'_1p' + a_{out}a_0 + a'_{out}a'_0p + a'_{out}a_1 + c'_1r'z' + a_{out}b_{out}z' + \\
 & b_{out}b'_1q' + b_{out}b_0 + b'_{out}b'_0q + b'_{out}b_1 + a_1b_{out}z' + a'_0b_{out}pz' + \\
 & c_{out}c'_1r' + c_{out}c_0 + c'_{out}c'_0r + c'_{out}c_1 + a'_0b_1pz' + a_1b_1z' + \\
 & b'_{out}s + a'_{out}s + a_{out}b_{out}s' + c_{out}t + c'_{out}t' + a_{out}b_1z' + c_0z' + \\
 & sz' + tz' + s't'z + a'_0b'_0pqz' + a_1b'_0qz' + a_{out}b'_0qz' + c'_{out}z' + \\
 & b'_1q't'z + a'_{out}t'z + b_0t'z + a_0t'z + a'_1p't'z + b'_{out}t'z + b'_{out}c'_0rz + \\
 & b'_{out}c_1z + b'_{out}c_{out}z + a'_1c'_0p'rz + a'_1c_1p'z + a'_1c_{out}p'z + a_0c'_0rz + \\
 & a_0c_1z + a_0c_{out}z + b_0c'_0rz + b_0c_1z + b_0c_{out}z + a'_{out}c'_0rz + \\
 & a'_{out}c_1z + a'_{out}c_{out}z + b'_1c'_0q'rz + b'_1c_1q'z + b'_1c_{out}q'z + c'_0rs'z + \\
 & c_1s'z + c_{out}s'z + b'_1q's + b_0s + a_0s + a'_1p's + a_{out}b'_0qs' + a_1b'_0qs' + \\
 & a'_0b'_0pqz' + c_0t' + a_{out}b_1s' + a_1b_1s' + a'_0b_1ps' + a'_0b_{out}ps' + \\
 & a_1b_{out}s' + c'_1r't' + c'_0rt + c_1t + a_1a_0 + b_1b_0 + c_1c_0 = 0.
 \end{aligned} \tag{4.19}$$

The variables in equation (4.20) which can be eliminated are a_{out} , b_{out} , c_{out} , s , and t . Since equation (4.20) is in Blake canonical form, terms which include literals of these variables are removed from the equation. Thus, the characteristic equation $\Phi_0(\underline{x}, \underline{y}, z) = 0$ of the circuit is

$$\begin{aligned}
 & c'_1r'z' + a'_0b_1pz' + a_1b_1z' + c_0z' + a'_0b'_0pqz' + a_1b'_0qz' + \\
 & a'_1c'_0p'rz + a'_1c_1p'z + a_0c'_0rz + a_0c_1z + b_0c'_0rz + b_0c_1z + \\
 & b'_1c'_0q'rz + b'_1c_1q'z + a_1a_0 + b_1b_0 + c_1c_0 = 0,
 \end{aligned} \tag{4.20}$$

where

- \underline{x} denotes the input variables p, q , and r , and
- \underline{y} represents the checkpoint variables a_0, a_1, b_0, b_1, c_0 , and c_1 . \square

Generation of Effective Test Vectors

The goal of test generation in a diagnostic system is to produce effective test vectors. An effective test vector is one for which the corresponding output cannot be deduced a priori. Only in this case can new information be derived from a test. To determine if an input vector is effective, we need to derive an expression in terms of the circuit inputs and output which demonstrates that the output is not uniquely deducible prior to a test.

Since a function with respect to the inputs and output of the circuit must be derived, the checkpoint variables can be eliminated from the characteristic function. The conjunctive eliminant of the characteristic function with respect to the checkpoint variables forms a new function, $\Theta(\underline{x}, z)$, which is a function of the circuit inputs and output. $\Theta(\underline{x}, z)$ is formed by the equation

$$\Theta(\underline{x}, z) = ECON(\Phi(\underline{x}, \underline{y}, z), \underline{y}). \quad (4.21)$$

The constraint $\Theta(\underline{x}, z) = 0$ holds by equation (B.62).

Using the Boolean expansion theorem, equation (B.38), $\Theta(\underline{x}, z)$ can be expanded in the form $\Theta(\underline{x}, z) = 0$ where Θ is defined by

$$\Theta(\underline{x}, z) = a(\underline{x})z' + b(\underline{x})z. \quad (4.22)$$

By consensus, equation (B.24), this equation may be rewritten as

$$a(\underline{x})z' + b(\underline{x})z + a(\underline{x})b(\underline{x}) = 0. \quad (4.23)$$

Using the property demonstrated by equation (B.32), this equation can be used to form three separate equations, i.e.,

$$a(\underline{x})b(\underline{x}) = 0 \quad (4.24)$$

$$a(\underline{x})z' = 0 \quad (4.25)$$

$$b(\underline{x})z = 0. \quad (4.26)$$

Equations (4.25) and (4.26) and the definition of the inclusion relation are used to generate the Boolean inclusions

$$a(\underline{x}) \leq z \quad (4.27)$$

$$z \leq b'(\underline{x}). \quad (4.28)$$

A range of values for z is depicted by

$$a(\underline{x}) \leq z \leq b'(\underline{x}). \quad (4.29)$$

Under the condition

$$a(\underline{x}) = b'(\underline{x}), \quad (4.30)$$

the state of the output z is fixed. Hence, the necessary condition to insure that z takes a range of values, i.e., is not deducible, is

$$a(\underline{x}) \neq b'(\underline{x}). \quad (4.31)$$

Since $a(\underline{x})$ and $b(\underline{x})$ are functions in a two-variable Boolean algebra, $a(\underline{x}) \neq b'(\underline{x})$ is equivalent to

$$a(\underline{x}) = b(\underline{x}). \quad (4.32)$$

By equation (B.30), equation (4.32) can be rewritten as

$$a'(\underline{x})b(\underline{x}) + a(\underline{x})b'(\underline{x}) = 0. \quad (4.33)$$

This equation and (4.24) can be combined by the property illustrated by equation (B.32) to form the equation

$$a'(\underline{x})b(\underline{x}) + a(\underline{x})b'(\underline{x}) + a(\underline{x})b(\underline{x}) = 0. \quad (4.34)$$

By consensus and absorption, the equation

$$a(\underline{x}) + b(\underline{x}) = 0 \quad (4.35)$$

is derived. A simple way to form $a(\underline{x}) + b(\underline{x})$ is to use the disjunctive eliminant of $\Theta(\underline{x}, z)$ with respect to the output variable z . Hence,

$$a(\underline{x}) + b(\underline{x}) = EDIS(\Theta(\underline{x}, z), z). \quad (4.36)$$

We will denote the function $a(\underline{x}) + b(\underline{x})$ as the input function $i(\underline{x})$. By equation (4.35),

$$i(\underline{x}) = 0. \quad (4.37)$$

Solutions of equation (4.37) are effective test vectors. The method outlined in Appendix B can be used to solve the equation. By equation (B.56)

$$i'(\underline{x}) = 1. \quad (4.38)$$

Each minterm, $m_j(\underline{x})$, where j is the decimal integer of the binary code for a minterm,² of the function $i'(\underline{x})$ is an effective test vector. Typically, a function has several minterms; hence, a number

²See Appendix B. Minterm Canonical Form.

of effective test vectors will be generated. Without creating an excessive amount of overhead, there exists no conceivable way to determine which minterm is the "best" test vector. The best test vector is one which would guarantee that an experiment will have a minimum number of tests. In any case, selection of the minterm which guarantees a minimal test set may be impossible because the results of previous tests guide the generation of future inputs.³ We cannot foresee constraints imposed on the test generation process that will result from the current test. Development of heuristics which may guide this process is a subject for further research.

We will arbitrarily choose a test vector given by a minterm $m_j(\underline{x})$ of the function $i'(\underline{x})$. Since any minterm of $i'(\underline{x})$ is an effective test vector, and the diagnostic process guides test vector generation, the set of test vectors generated in the course of an experiment should be near-minimal. The next section will show how the result of a test guides the diagnostic process.

Deduction of New Information

A minterm $m_j(\underline{x})$ of $i'(\underline{x})$ is specified uniquely by the equation

$$m_j(\underline{x}) = 1 \quad (4.39)$$

for some j , where j is the decimal integer of the binary code for a minterm. When a test vector is applied to the circuit under test, the output z of the circuit will have the value $r \in \{0, 1\}$. This relationship is given by

$$z = r, \quad r \in \{0, 1\}. \quad (4.40)$$

Thus, a test yields the implication

$$m_j(\underline{x}) = 1 \Rightarrow z = r \quad r \in \{0, 1\}. \quad (4.41)$$

³This is discussed in the next section.

Equation 4.39 is equivalent to

$$m'_j(\underline{x}) = 0. \quad (4.42)$$

By (B.30), equation (4.40) can be placed in the form

$$z \oplus r = 0. \quad (4.43)$$

Using equations (4.42) and (4.43), equation (4.41) can be rewritten as

$$m'_j(\underline{x}) = 0 \Rightarrow z \oplus r = 0. \quad (4.44)$$

By the Extended Verification Theorem, (4.44) is equivalent to the inclusion

$$z \oplus r \leq m'_j(\underline{x}). \quad (4.45)$$

Using the definition of the inclusion relation, (4.45) can be represented by the equation

$$m_j(\underline{x}) \cdot (z \oplus r) = 0. \quad (4.46)$$

This equation represents information found by a test. This information places constraints on the characteristic equation which guide the test vector generation process. Depending on the circuit response r to the test, one of two constraints can be added to the characteristic equation. These are given by

$$m_j(\underline{x}) \cdot z = 0 \quad (r = 0), \quad (4.47)$$

$$m_j(\underline{x}) \cdot z' = 0 \quad (r = 1). \quad (4.48)$$

The constraint formed by the i th test is given by the equation

$$m_{j(i)}(\underline{x}) \cdot (z \oplus r(i)) = 0, \quad (4.49)$$

where

- $m_{j(i)}(\underline{x})$ is the test vector, generated using the characteristic function $\Phi_{i-1}(\underline{x}, \underline{y}, z)$, and applied during the i th test, and
- $r(i)$ is the response of the circuit to the i th test.

After test i , equation (4.49) is combined with the characteristic equation $\Phi_{i-1}(\underline{x}, \underline{y}, z) = 0$ by (B.32) to form the equation

$$\Phi_{i-1}(\underline{x}, \underline{y}, z) + m_{j(i)}(\underline{x}) \cdot (z \oplus r(i)) = 0. \quad (4.50)$$

After equation (4.50) is formed, new conclusions can be derived. However, these results are not explicit in the equation. All such conclusions are revealed only by expressing the left-hand side of (4.50) in Blake canonical form. This form is given by

$$BCF(\Phi_{i-1}(\underline{x}, \underline{y}, z) + m_{j(i)}(\underline{x}) \cdot (z \oplus r(i))) \quad (4.51)$$

We will denote the formula given by (4.51) as $\phi(\underline{x}, \underline{y}, z)$. Terms of ϕ can be placed into three categories. These are:

1. Terms in which one of the literals is the output variable z ,
2. Terms in which one the literals is the complement of the output variable z' , and
3. Terms which do not include either form of the output variable.

Terms of $\phi(\underline{x}, \underline{y}, z)$ which include literals that are either input or output variables, but not checkpoint variables, are the terms that remain when $\Theta(\underline{x}, z)$ is found by conjunctive elimination of $\Phi_i(\underline{x}, \underline{y}, z)$ with respect to \underline{y} as defined by equation (4.21). These terms are significant because

they contain the information necessary to generate effective tests. If no such terms exist, i.e., if Θ is identically zero, then there exists no direct relationship between the input and output variables. In this circumstance, an arbitrary test vector is effective.

Terms of $\phi(\underline{x}, \underline{y}, z)$ which do not include the output variable z take one of two forms. Either these terms consist only of checkpoint variables, or they include input as well as checkpoint variables. By (B.62), conjunctive elimination of ϕ with respect to the output variable z yields a new equation

$$h(\underline{x}, \underline{y}) = 0 \quad (4.52)$$

consisting of terms which only include the input and checkpoint variables. The vector \underline{x} is independent; the values of the \underline{x} variables can be freely changed. The \underline{y} -variables are fixed at some value depending on the fault conditions that exist in the circuit. By (B.32), each term of $h(\underline{x}, \underline{y})$ also equals 0. Then, for each term an equation such as

$$a_0 b_1 x_1 = 0 \quad (4.53)$$

is formed, where

- a_0 and b_1 are checkpoint variables, and
- x_1 is an input variable.

Since x_1 is independent, we can assign it the value of 1. Then, (4.53) remains an identity only if

$$a_0 b_1 = 0. \quad (4.54)$$

Given this property, the input variables can be deleted from the terms of (4.52). The disjunctive eliminant of $h(\underline{x}, \underline{y})$ with respect to \underline{x} is a new function $g(\underline{y})$; terms of $g(\underline{y})$ only involve checkpoint variables. The function $g(\underline{y})$ is used to form the equation

$$g(\underline{y}) = 0 \quad (4.55)$$

By (B.32), $\phi(\underline{x}, \underline{y}, z)$ and (4.55) can be combined to form a new function

$$\phi(\underline{x}, \underline{y}, z) + g(\underline{y}). \quad (4.56)$$

Absorption is used to further simplify this function. After simplification we define the function to be in *diagnostic canonical form*. In summary, the diagnostic canonical form is derived as follows:

1. Add the constraint (4.49) generated by the i th test to the characteristic function $\Phi_{i-1}(\underline{x}, \underline{y}, z)$ to form a new function $\phi(\underline{x}, \underline{y}, z)$,
2. Obtain the Blake canonical form of the newly-formed function $\phi(\underline{x}, \underline{y}, z)$,
3. Remove the input variables from the terms of $\phi(\underline{x}, \underline{y}, z)$ which include only input and checkpoint variables to form $g(\underline{y})$,
4. Add $g(\underline{y})$ to the newly-formed function $\phi(\underline{x}, \underline{y}, z)$, and
5. Perform absorption to remove superfluous terms. The resulting function is in Blake canonical form.

The resulting function in diagnostic canonical form is the updated characteristic function $\Phi_i(\underline{x}, \underline{y}, z)$.

Formation of the diagnostic canonical form is significant because it enables the efficient deduction of relationships among variables of the characteristic function.

Combining steps outlined previously, the function $g(\underline{y})$ is derived as follows:

$$g(\underline{y}) = EDIS(ECON(\phi(\underline{x}, \underline{y}, z), z), \underline{x}) \quad (4.57)$$

The function $g(\underline{y})$ is used to augment the function $\phi(\underline{x}, \underline{y}, z)$ to form the diagnostic canonical form. After simplification by absorption, it is guaranteed that further augmentations will produce no new result. This is demonstrated by the following theorem and proof.

Theorem 4.1. Given a Boolean function $\Phi(\underline{x}, \underline{y}, z)$, define a Boolean function AUG as follows:

$$AUG(\Phi(\underline{x}, \underline{y}, z)) = \Phi(\underline{x}, \underline{y}, z) + EDIS(ECON(\Phi(\underline{x}, \underline{y}, z), z), \underline{x}). \quad (4.58)$$

Then

$$AUG(AUG(\Phi(\underline{x}, \underline{y}, z))) = AUG(\Phi(\underline{x}, \underline{y}, z)). \quad (4.59)$$

Proof.

By (B.57),

$$ECON(AUG(\Phi), z) = (AUG(\Phi(\underline{x}, \underline{y}, 0)) \cdot (AUG(\Phi(\underline{x}, \underline{y}, 1))) \quad (4.60)$$

$$\begin{aligned} &= [\Phi(\underline{x}, \underline{y}, 0) + EDIS(ECON(\Phi, z), \underline{x})] \cdot \\ &\quad [\Phi(\underline{x}, \underline{y}, 1) + EDIS(ECON(\Phi, z), \underline{x})] \\ &= (\Phi(\underline{x}, \underline{y}, 0)) \cdot (\Phi(\underline{x}, \underline{y}, 1)) + EDIS(ECON(\Phi, z), \underline{x}) \\ &= ECON(\Phi, z) + EDIS(ECON(\Phi, z), \underline{x}) \end{aligned} \quad (4.61)$$

Since $\mathcal{F} \leq EDIS(\mathcal{F}, x) \ \forall \mathcal{F}, x$, the last equation of (4.60) reduces to

$$ECON(AUG(\Phi), z) = EDIS(ECON(\Phi, z), \underline{x}) \quad (4.62)$$

Using this result,

$$AUG(AUG(\Phi)) = AUG(\Phi) + EDIS(ECON(AUG(\Phi), z), \underline{x}) \quad (4.63)$$

$$= AUG(\Phi) + EDIS(EDIS(ECON(\Phi, z), \underline{x}), \underline{x}) \quad (4.64)$$

Since $EDIS(ECON(\Phi, z), \underline{x})$ yields a function which is independent of \underline{x} , the disjunctive eliminant of this new function with respect to \underline{x} yields the same function. Hence,

$$AUG(AUG(\Phi)) = AUG(\Phi) + EDIS(ECON(\Phi, z), \underline{x}). \quad (4.65)$$

Expanding $AUG(\Phi)$ yields

$$AUG(AUG(\Phi)) = [\Phi + EDIS(ECON(\Phi, \underline{x})] + EDIS(ECON(\Phi, z), \underline{x}), \quad (4.66)$$

which is the same as

$$AUG(AUG(\Phi)) = \Phi + EDIS(ECON(\Phi, z), \underline{x}) \quad (4.67)$$

$$= AUG(\Phi) \quad (4.68)$$

Q.E.D. [Brown 88b]

One property of the updated characteristic function, $\Phi_i(\underline{x}, \underline{y}, z)$, is that it is larger than the previous characteristic function, $\Phi_{i-1}(\underline{x}, \underline{y}, z)$. This is denoted by

$$\Phi_{i-1}(\underline{x}, \underline{y}, z) \leq \Phi_i(\underline{x}, \underline{y}, z). \quad (4.69)$$

The updated characteristic function $\Phi_i(\underline{x}, \underline{y}, z)$ is used to generate the input function $i_{i+1}(\underline{x})$; the input function is used to generate the test vector $m_{j(i+1)}(\underline{x})$. Because the input function is generated from the updated characteristic function, as the characteristic function gets larger, the corresponding input function also gets larger. Hence.

$$i_i(\underline{x}) \leq i_{i+1}(\underline{x}) \quad (4.70)$$

Essentially, there exists a search space for effective inputs. Information discovered from a test acts to further limit this search space at each iteration. Eventually, enough information will have been gained through testing to cover the entire search space. At this point further effective inputs do not exist; hence, all information deducible from testing is obtained. Since the input function gets larger after each iteration, it will eventually become

$$i_{i+1}(\underline{x}) = 1. \quad (4.71)$$

When this occurs, all information regarding the state of the checkpoint variables that is deducible through testing, i.e., the possible faults in the circuit, as well as the relationship between the input and output variables, i.e., the function that the faulty circuit is performing, is implicit in $\Phi_i(\underline{x}, \underline{y}, z)$. The final characteristic function will be denoted $\Phi_n(\underline{x}, \underline{y}, z)$, i.e., the characteristic function after the n th test. The final input function is denoted $i_{n+1}(\underline{x})$. The function $\Phi_n(\underline{x}, \underline{y}, z)$ must be manipulated to determine the fault conditions in the circuit as well as the function that the faulty circuit is performing.

Interpretation of Results

Once all information deducible by testing has been obtained, the characteristic function that exists at that point, $\Phi_n(\underline{x}, \underline{y}, z)$, incorporates the possible states of the checkpoint variables in the circuit as well as the function that the faulty circuit is performing. To derive this information, we need to produce an equation which expresses the circuit output as a function of the inputs and an equation which represents the possible states of the checkpoint variables.

Derivation of the Circuit Function. Restating equation (4.22) in the discussion of effective test vector generation, $\Theta(\underline{x}, z)$ can be expanded

$$\Theta(\underline{x}, z) = a(\underline{x})z' + b(\underline{x})z. \quad (4.72)$$

The necessary condition which insures that z is not deducible was given by (4.31). This condition is

$$a(\underline{x}) \neq b'(\underline{x}). \quad (4.73)$$

Thus, the condition for which z is deducible, i.e., information cannot be obtained from testing, is given by

$$a(\underline{x}) = b'(\underline{x}). \quad (4.74)$$

Equation (4.74) is the equation obtained when the input function $i(\underline{x})$ becomes equal to 1. This is shown as follows. By equations (4.36), (4.37), and (4.72):

$$\begin{aligned} i(\underline{x}) &= EDIS(\Theta(\underline{x}, z), z) \\ &= EDIS((a(\underline{x})z' + b(\underline{x})z), z). \end{aligned} \quad (4.75)$$

By the definition of the disjunctive eliminant,

$$\begin{aligned} i(\underline{x}) &= \sum_{z \in \{0,1\}} (a(\underline{x})z' + b(\underline{x})z) \\ &= a(\underline{x}) + b(\underline{x}). \end{aligned} \quad (4.76)$$

It was determined in the previous section that all information deducible from testing was obtained when the input function $i(\underline{x})$ becomes equal to 1. Hence, at the completion of testing, the following condition is true

$$a(\underline{x}) + b(\underline{x}) = 1. \quad (4.77)$$

Combining (4.77) with the fact that $a(\underline{x})b(\underline{x}) = 0$ by equation (4.24), the definition of complementation implies that the following condition must be true

$$a(\underline{x}) = b'(\underline{x}), \quad (4.78)$$

or alternatively,

$$a'(\underline{x}) = b(\underline{x}). \quad (4.79)$$

This result confirms equation (4.74). At the completion of testing, equation (4.22) can then be restated as

$$a(\underline{x})z' + a'(\underline{x})z = 0. \quad (4.80)$$

By (B.30), this equation can be rewritten as

$$a(\underline{x}) \oplus z = 0. \quad (4.81)$$

Equivalently,

$$a(\underline{x}) = z. \quad (4.82)$$

This equation yields the output of the circuit as a function of the circuit inputs. This is the function that the potentially faulty circuit is performing. In summary, $a(\underline{x})$ is found as follows:

1. Determine $\Theta(\underline{x}, z)$ by $ECON(\Phi_n(\underline{x}, \underline{y}, z), \underline{y})$, and
2. Set $z = 0$ in $\Theta(\underline{x}, z)$ to obtain $a(\underline{x})$.

The function $a(\underline{x})$ that exists at the completion of testing will be denoted the fault function and designated as $F(\underline{x})$. The fault function is used to determine the possible states of the checkpoint variables.

Determination of Possible Checkpoint States. As in the case of the fault function, a function which yields information concerning the state of the checkpoint variables must be derived. The constraints on the checkpoint variables were demonstrated by equation (4.11). The left-hand side of this equation is the constraint function, designated as $g(\underline{y})$. Equation (4.9) is an equation which expresses the relationships among the circuit inputs, output, and checkpoint variables. With some manipulation, this equation could be placed in the form

$$z = f(\underline{x}, \underline{y}), \quad (4.83)$$

where

- \underline{x} are the circuit inputs,
- \underline{y} are the checkpoint variables, and
- z is the circuit output.

For all inputs, the fault function $F(\underline{x})$ is equivalent to the function $f(\underline{x}, \underline{y})$. However, the possible values of the vector \underline{y} are undetermined. If all inputs were to be exhaustively applied to the circuit, then the following system of equations would be obtained

$$\begin{aligned} F(0, 0, \dots, 0) &= f(0, 0, \dots, 0, \underline{y}) \\ F(0, 0, \dots, 1) &= f(0, 0, \dots, 1, \underline{y}) \\ &\vdots \\ F(1, 1, \dots, 1) &= f(1, 1, \dots, 1, \underline{y}). \end{aligned} \quad (4.84)$$

Using reduction, this system of equations may be rewritten as

$$\sum_{\underline{a} \in \{0,1\}^n} F(\underline{a}) \oplus f(\underline{a}, \underline{y}) = 0. \quad (4.85)$$

By equation (B.32), $g(\underline{y})$ can be combined with (4.85) to form the equation

$$g(\underline{y}) + \sum_{\underline{a} \in \{0,1\}^n} F(\underline{a}) \oplus f(\underline{a}, \underline{y}) = 0. \quad (4.86)$$

[Brown 79:12]

Using the definition of the disjunctive eliminant, (4.86) may be rewritten as

$$g(\underline{y}) + EDIS((F(\underline{x}) \oplus f(\underline{x}, \underline{y})), \underline{x}) = 0. \quad (4.87)$$

[Brown 79:11]

The left-hand side of this equation yields a function which incorporates the possible states of the checkpoint variables. This function will be denoted the checkpoint state function and designated as $G(\underline{y})$. Once the fault function $F(\underline{x})$ is found using the results of the previous section, $G(\underline{y})$ can be determined.

Solutions of the equation

$$G(\underline{y}) = 0 \quad (4.88)$$

yield the possible states of the checkpoint variables. Typically, the states of all checkpoints cannot be positively determined. This is because combinations of faults within the same equivalence class cause a circuit to behave identically. Thus, the set of solutions for (4.88) represents the possible faults that may exist in the circuit under test. There exists no way to distinguish between the fault combinations; however, exactly one solution of (4.88) represents the faults that exist in the circuit. The method found in Appendix B may be used to obtain solutions. Using this method, the following equation is obtained:

$$G'(\underline{y}) = 1 \quad (4.89)$$

Minterms of $G'(\underline{y})$ are the possible node states. In some cases a literal may appear in all minterms of $G'(\underline{y})$. In these cases, information pertaining to the checkpoint node associated with the literal is attained with certainty. Using the circuit of Figure 4.5, if the literal y_{10} appeared in uncomplemented form in all solutions of $G'(\underline{y})$, then it could be stated with certainty that the node x_1 is stuck-at-0 (by Table 4.1). All minterms can be examined to determine the nodes for which the state is certain; the remaining literals in each minterm yield the possible states of the checkpoint nodes in the circuit.

Applications of the Diagnostic Algorithm

The procedure described in this chapter is intended for complete diagnosis of faults within a single-output combinational circuit. However, there are a number of other uses for the procedure. These include generation of fault detection test sets, diagnosis of specific points in the circuit, and

using the algorithm to evaluate a test set. Implementation of all of these applications is a subject for future work. In this project, however, a full diagnostic system is designed and implemented.

The diagnostic algorithm can be used to generate test sets for fault detection experiments. To use the procedure in this way, after a test is constructed, the fault-free circuit's response to the test is determined. Then, this output is fed back to the procedure using equation (4.49). The algorithm continues to generate tests until the correct outputs provide enough information to validate the possible states of the checkpoint variables. Since only effective tests are generated and the correct outputs guide the production of these tests, the resulting test set would be near-minimal. The test set is not guaranteed to be minimal because at each iteration of the test generation process, a test is selected arbitrarily from the solutions to equation (4.37).

A by-product of test set generation is the creation of a list of undetectable, i.e., redundant, faults in the circuit for which the test set is generated. If the correct output always is fed back to the procedure in (4.49), the circuit would be functioning normally. By definition, redundant faults do not cause a circuit to act abnormally. The diagnostic procedure always produces a listing of the faults that may exist in the circuit. Hence, the procedure produces a list of the undetectable faults which may occur in spite of the fact that the correct outputs are fed back. This information may be helpful early in the design process, because redundant faults occur due to redundancies in a circuit.

In the derivation of the characteristic function of the circuit, checkpoints were used to denote potential locations of faults. This derivation assumed that the characteristic function is to be used for multiple-fault location experiments. A modification of the characteristic function would be to allow arbitrary points to be designated as checkpoints. The characteristic function would be developed based on these user-designated checkpoints. The diagnostic procedure would generate tests required to determine the states of the user-designated checkpoints. Thus, tests can be generated to detect the existence of specific faults in a circuit.

Another application of the diagnostic algorithm is to evaluate a pre-computed test set. For each test in the test set, the fault-free circuit's response may be computed. Using (4.49), a test and response are added to the characteristic function of a circuit. After generation of the updated characteristic function, another test and response are added. After all tests and responses are exhausted, the characteristic function would be interpreted as described in the Interpretation of Results section. A list of possible faults in the circuit is produced; these faults are the faults that are not detected by the test set being evaluated. Additionally, after all test/response pairs are processed, the input function $i_{i+1}(x)$ may be formed. If the input function is not identically equal to 1 by equation (4.71), then effective test vectors exist that were not a member of the pre-computed test set. Hence, the diagnostic algorithm can show whether or not a pre-computed test set is a complete fault-detection test set.

Advantages and Limitations of the Diagnostic Algorithm

The diagnostic algorithm meets many of the criteria defining an ideal diagnostic system. In doing so, the algorithm overcomes many of the limitations which restrict the capability of other techniques. Such restrictions include the single-fault assumption, requirements to generate a fault detection test set prior to an adaptive experiment, and the necessity of determining the probabilities of possible faults.

The algorithm presented was developed specifically to adaptively locate multiple stuck-at faults in combinational circuits. Additionally, as described in the previous section, this algorithm also can be used to generate fault detection test sets for preset fault detection experiments. Whether used for adaptive fault location or test set generation, the use of constraints to guide the test vector generation process insures that a near-minimal test set is generated.

Processing which is required in other diagnostic techniques is avoided in this algorithm. A priori fault enumeration, required in well-known methods such as the D algorithm, is not required.

Some diagnostic systems transform a circuit representation into an equivalent circuit representation for which test vectors are generated; tests which detect faults in the equivalent circuit detect faults in the actual representation of the circuit. The method developed in this chapter does not necessitate a transformation to an equivalent circuit. Furthermore, because the generation of test vectors to detect multiple faults is implicit in the procedure, the diagnostic algorithm does not require performance of a masking analysis. A masking analysis is used in some techniques which generate tests to detect single faults, perform a masking analysis to determine faults not covered by the single-fault detection test set, and then generate tests to cover faults that were not detected by the initial test set.

As previously discussed, the model used in the diagnostic algorithm imposes limitations which prevent the algorithm from meeting all of the criteria of an ideal diagnostic system. Describing the circuit with Boolean equations at the gate level necessitates diagnosis of faults at the gate level. Additionally, the checkpoint model used in this procedure is predicated on the stuck-at fault model; hence, non-classical faults cannot be diagnosed with the diagnostic algorithm. A restriction imposed to narrow the scope of the project was to develop the algorithm for single-output combinational circuits; however, the algorithm imposes no other restrictions on circuit topology. Likewise, there are no limitations on the type of logic gate that can be used by a circuit diagnosed using this algorithm. Extension of the diagnostic algorithm to overcome these limitations is a subject for future work.

Another limitation of the diagnostic algorithm is that it cannot detect a bad model. Although the original circuit representation may be erroneous, the algorithm will generate tests regardless of this fact. Of the methods surveyed in Chapter 2, only Davis's method of diagnosis using structure and behavior can determine that a model is incorrect. However, Davis's method is limited by the single-fault assumption.

In the following chapters, the design and implementation of the diagnostic algorithm developed in this chapter is discussed. Included in the discussion are implementation decisions which impose minor limitations on the structure of a circuit. However, before implementation issues are examined, the next chapter describes the architecture of the diagnostic system.

V. Architecture of the Diagnostic System

In the previous chapter the mathematical basis of the diagnostic system was developed. In this chapter, an architecture is developed to serve as a basis for a software implementation of the system. There are two important characteristics of a software architecture: partitioning of the system into a hierarchical structure of modules, and definition of data within the system [Press 87:218]. A problem must be partitioned into units that can be designed and easily maintained. In Pressman's book on software engineering, he cites that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Press 87:222]. Data definition is important because this defines the interrelationships between the modules.

The key to effectively partitioning the diagnostic system is the ability to decompose the algorithm developed in the previous chapter into separate components. Fortunately, the algorithm is actually a series of steps which proceeds from a system of equations which describes the circuit, to derivation of the characteristic equation of the circuit, generation of effective test vectors, deduction of new information from results of a test, and finally to the interpretation of the results of the input-output experiment. This stepwise construction of the algorithm facilitates easy partitioning into independent modules. Thus, the architecture specified in this chapter is a functional decomposition of the diagnostic algorithm developed previously.

Items which are defined in this architecture include the partitioning of the system into separate modules, descriptions of the function of each module, information that must be passed between modules, and user interfaces. The system is partitioned into four components: the input module, equation generation module, tester module, and interpretation module. Development of this architecture was an iterative process. After initial development, revised requirements often necessitated refinement of the original architecture. The architecture described in this chapter is the version which resulted after full development of the diagnostic system.

The Input Module

A system of Boolean equations can be used to describe a combinational circuit. Another way of describing a combinational circuit is by a VHDL description. To give the user flexibility, either form of circuit description may be input to the diagnostic system. To facilitate the use of different types of descriptions, the descriptions are converted to a common format which retains information about the structure of the circuit. This common form, which is used by the remaining modules of the diagnostic system, will be denoted the intermediate format. The first module in the system, the input module, will convert either of the two input descriptions to the intermediate format. Figure 5.1 is a pictorial description of the module.

The input module includes a dual set of operations. One set of operations converts the Boolean equations that describe the circuit into the intermediate format; the complementary set of operations transforms the VHDL description to the intermediate format. In each case, the input description will be input to the system from a data file. The input module prompts the user to determine the type of input description that will be used. When a file is read, a tokenizer tailored to the specific type of description converts the characters in the file to a list of symbols, or tokens. The list of tokens is transformed to the intermediate format by an appropriate parser. The intermediate format will be an equation-like form which meets the requirements of the mathematics developed in Chapter 4. However, the exact specification of the intermediate format is implementation-dependent and will be discussed in the next chapter. The formats for the Boolean equations and VHDL descriptions are defined in the succeeding sections.

Boolean Equation Format. The equation format accepted by the diagnostic system is a system of Boolean equations. To devise this set of equations, each node in a circuit must be labeled by a unique identifier or label as shown in Figure 5.2. Each logic gate in the circuit is denoted by a single equation; each line of the input file contains an equation representing a gate. Thus, if there

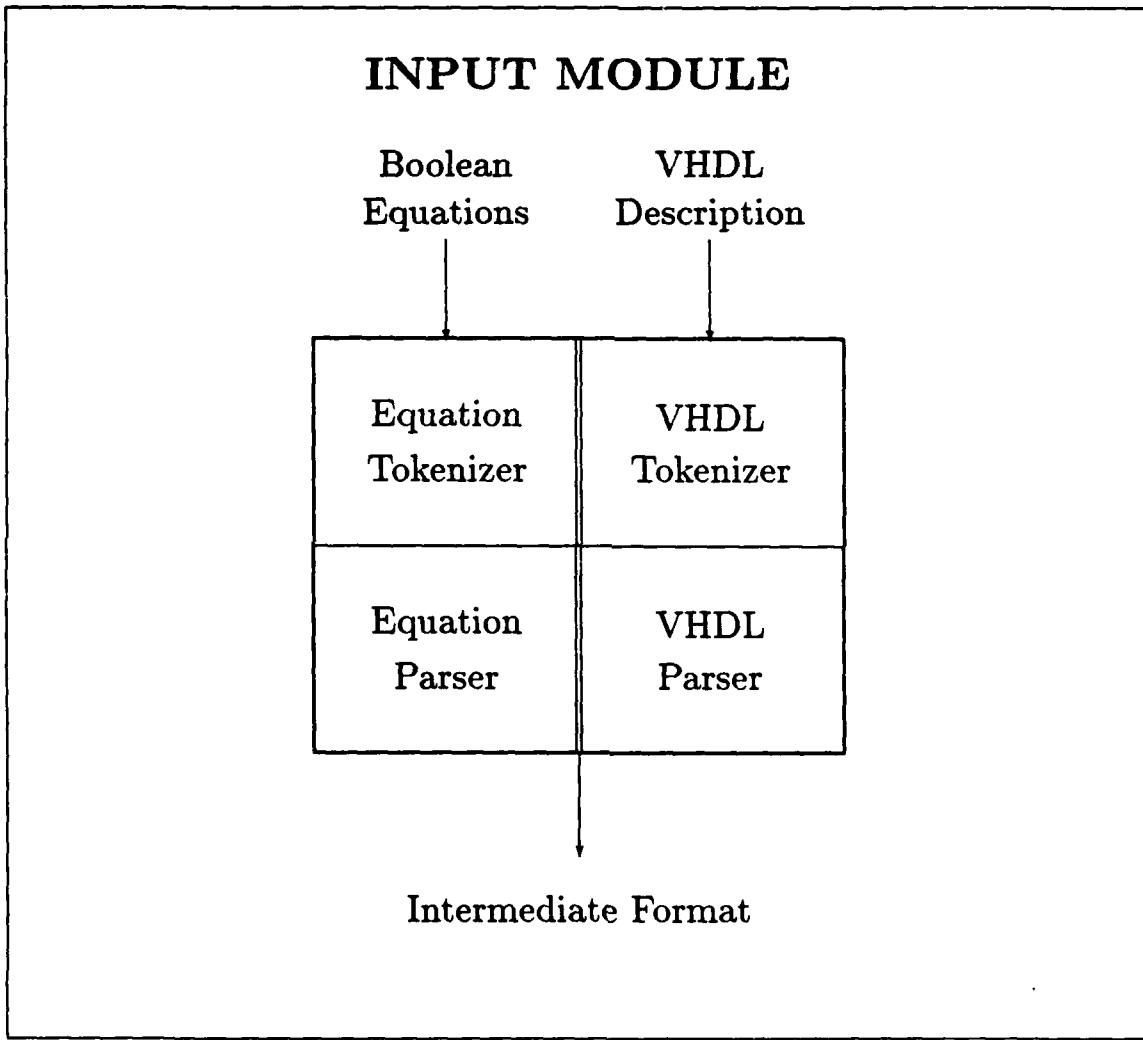


Figure 5.1. The Input Module

are four gates in a circuit, the input file that describes the circuit would contain four lines—one equation per line. Each equation must be of the form

```
output = input operator input
```

where the output node of the gate is always on the left side of the equals sign, and the right side of the equals sign contains the gate inputs and the operators specific to the type of gate.

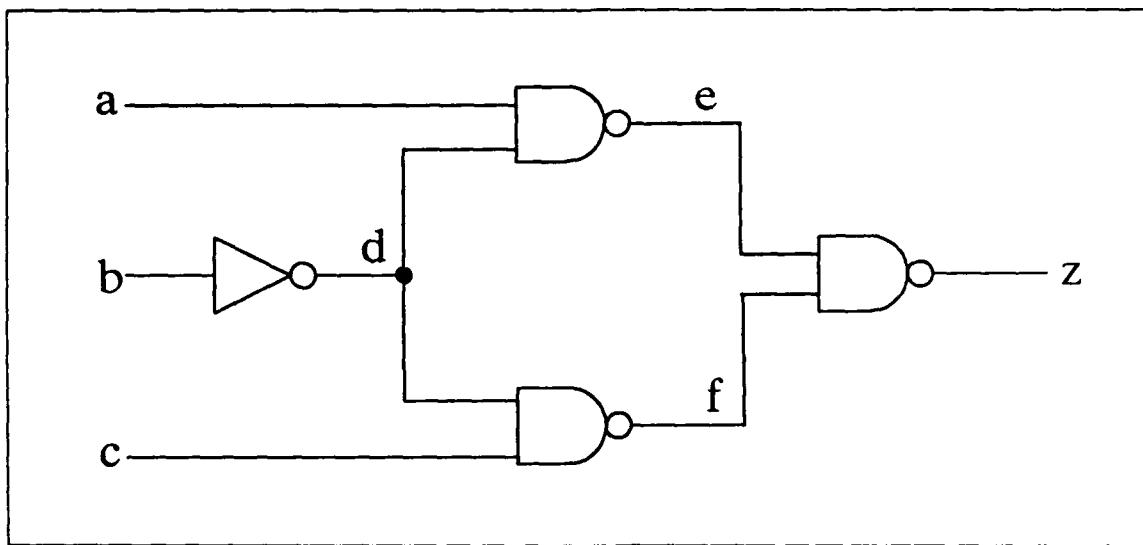


Figure 5.2. Circuit Demonstrating Boolean Equation Format

AND Gate Representation. An AND gate can be represented in several different ways. The * operator is used to denote the AND operation. One representation of an AND gate is

```
output = first_input * second_input.
```

For a multiple-input AND gate, a string of inputs and operators is combined in the following way:

```
output = first_input * second_input * third_input.
```

Alternate ways of representing an AND gate are to enclose the inputs and operators with parentheses and to drop the operators entirely. The AND operation is assumed when an operator is replaced by the juxtaposition of input symbols. Examples of these forms are

```
output = (first_input * second_input * third_input)
output = (first_input second_input third_input)
output = first_input second_input third_input.
```

OR Gate Representation. An OR gate also can be represented in several different ways. The + operator is used to denote the OR operation. The representation of an OR gate is

```
output = first_input + second_input.
```

For a multiple-input OR gate, a string of inputs and operators is combined, i.e.,

```
output = first_input + second_input + third_input.
```

An alternate way of representing an OR gate is to enclose the inputs and operators with parentheses.

An example of this form is

```
output = (first_input + second_input + third_input).
```

NOT Gate Representation. A NOT gate (Inverter) is represented in two different manners. The ' operator is used to denote the NOT operation. The first representation of a NOT gate is

```
output = input'.
```

The alternate representation of a NOT gate is to enclose the input in parentheses. An example of this form is

```
output = (input)'.
```

XOR Gate Representation. An XOR (Exclusive-OR) gate is represented similarly to an OR gate, except that the + operator is replaced by the ! operator. The ! symbol was chosen because the \oplus symbol that normally represents the XOR operation is unavailable on a computer keyboard, and the ! symbol is symmetric like the \oplus operator. Examples of equation which represent XOR gates are

```
    output = first_input ! second_input
    output = first_input ! second_input ! third_input
    output = (first_input ! second_input ! third_input).
```

NAND Gate Representation. A NAND gate cannot by represented in the same manner as AND, OR, or XOR gates, because the NAND operation is not associative. One view of the NAND gate is that it is a complemented AND or NOT-AND gate. Thus, the AND and NOT operators can be combined to form a NAND gate. In this representation, the AND portion of the equation must be enclosed in parentheses; the NOT operator is placed outside the parentheses. A representation of a two-input NAND gate is

```
    output = (first_input * second_input)'.
```

For a multiple-input NAND gate, a string of inputs and operators is combined in the following way:

```
    output = (first_input * second_input * third_input)'.
```

Juxtaposition may also be used to represent a NAND gate:

```
    output = (first_input second_input third_input)'.
```

NOR Gate Representation. Like the NAND operation, the NOR operator is not associative. Thus, it must be denoted in a similar manner. The NOR operation is represented by a combination of the OR and NOT operators. Parentheses enclose the OR portion of the operation;

the NOT operator is positioned outside of the parentheses. A representation of a two-input NOR gate is

```
output = (first_input + second_input)'.
```

For a multiple-input NOR gate, a string of inputs and operators is combined in the following way:

```
output = (first_input + second_input + third_input)'.
```

XNOR Gate Representation. An XNOR (Exclusive NOR) gate is represented similarly to the NAND and NOR gates. A combination of XOR and NOT operators denotes an XNOR operation. Examples of ways to represent XNOR gates are:

```
output = (first_input ! second_input)'
output = (first_input ! second_input ! third_input)'.
```

Using these gate definitions, the equations for the circuit given by Figure 5.2 are given in Figure 5.3.

$d = b'$	$d = (b)'$
$e = (a \cdot d)'$	$e = (a * d)'$
$f = (c \cdot d)'$	$f = (c * d)'$
$z = (e \cdot f)'$	$z = (e * f)'$

or

Figure 5.3. Equations Representing Circuit of Figure 5.2

Definition of VHDL Input Format. The alternate method for representing a circuit is by a VHDL description. Representing a design with VHDL is advantageous for two reasons. The act of describing a system forces the designer to think out a design clearly; also, the VHDL description can be used to simulate the hardware design. A “*design entity*” is the primary hardware abstraction in VHDL” [IEEE 88:1-1]. Any part of a design which has a specified function and interface can be

represented as a design entity [IEEE 88:1-1]. This entity can represent an entire design, a circuit, subcircuit, or a specific component. Typically, components are interconnected to form a design hierarchy of hardware elements. Thus, a hardware design is described and constructed in a modular fashion for the same reasons that software architectures are partitioned. Each design entity in VHDL is defined by two items, an *entity declaration* and an *architecture body* [IEEE 88:1-1]. The *entity declaration*¹ is the portion of a design entity which describes the interface between a design entity and the environment in which it is instantiated [IEEE 88:1-1]. The *architecture body* describes the function of the design entity. Hence, the architecture body "specifies the relationships between the inputs and outputs of a design entity" [IEEE 88:1-6].

The architecture body can be described in either behavioral, structural, or dataflow forms [IEEE 88:1-6]. A *behavioral* view is normally a way of algorithmically describing the function of a circuit or component. This form would not necessarily embody the structure of a design, rather it describes how it acts. The *structural* view of VHDL describes a design entity by a collection of lower-level design entities. For example, a combinational logic circuit can be described in the structural view of VHDL by a collection of design entities representing the different types of gates that compose the circuit. Each of the gates would have its own associated entity declaration and architecture body. The *dataflow* view of VHDL describes a circuit or component by a collection of VHDL signal assignment statements. A *signal_assignment_statement* is of the form

```
target <= waveform
```

where **target** is a signal which receives the value of **waveform** [IEEE 88:8-3]. **Waveform** is composed of a collection of signals and operators which combine to form a value which is then assigned to **target**. Design entities may also be composed of a mixture of the three different views of component description.

¹Terms listed in the typewriter typeset are defined in Appendix C. Definition of VHDL Subset.

The behavioral view, because it may not represent the structure the circuit, is unsuited for describing a circuit to be diagnosed using the method developed in Chapter 4. Thus, this view will not be considered for the diagnostic system.

Either a structural or dataflow representation can describe the structure of a circuit. The structural representation is perhaps the best way of describing a complex design. The entity declarations and architectural bodies of a design are normally spread throughout several different files of a design library. Transforming circuits represented by a structural VHDL description into a form required for this diagnostic system would be a complex task. When each component is instantiated in the highest-level design entity, the file associated with the component would have to be found and transformed also. Furthermore, there could exist a nesting of component instantiations which would have to be handled by the system.

The dataflow representation can be used to define a simple circuit or component. The function of a logic gate can be described by a single signal assignment statement in its architecture body. A combinational logic circuit may be represented by a series of signal assignment statements, where a signal assignment statement represents a single gate in the circuit.

Because only combinational circuits will be diagnosed by this system, the use of a complex form of VHDL description is not required. Thus, to narrow the scope of this project, a dataflow representation will be used for circuit description. This representation constrains the user to what is known as a "flat" description of a circuit. This form of description would be inappropriate for circuits any more complex than combinational circuits. However, in an early implementation of VHDL, there existed a tool called a simplifier which converted hierarchical design descriptions into a single flat representation [Inter 87]. The form of this flat description was under the user's control. Such a tool could allow the conversion of structural VHDL descriptions to the dataflow representation required by this diagnostic system. Extension of the diagnostic system to allow the

use of structural VHDL and construction of a simplifier for current implementations of the description language are subjects for further study.

VHDL is currently defined by ANSI/IEEE Standard 1076-1987 [IEEE 88]. A subset of this standard must be clearly specified to insure that VHDL circuit descriptions which will be input to the diagnostic system conform to its requirements. This subset is defined in Appendix C. The intent of this subset is to describe a combinational circuit by a single VHDL architecture and a gate by a single signal assignment statement. The subset defined in Appendix C includes all of the elements of VHDL required to construct both entity declarations and architecture bodies. Circuits constructed using this subset will simulate properly using an IEEE Standard 1076-version VHDL analyzer and simulator. Entity declarations and architecture bodies will be assumed by the diagnostic system to be in separate files. The only part of a design entity that is used by the diagnostic system is the architecture body.

Entity Declaration Format. The entity declaration of a circuit to be diagnosed by the diagnostic system is required only to simulate the circuit with a VHDL analyzer and simulator. The form of the entity declaration is restricted by the subset of VHDL specified in Appendix C. An example entity declaration for the circuit in Figure 5.2 is given in Figure 5.4.

```
-- entity declaration for the circuit of Figure 5.2
entity CircuitFigure5.2 is
    port(A, B, C : in bit; Z : out bit);
end CircuitFigure5.2;
```

Figure 5.4. VHDL Entity Declaration of the Circuit in Figure 5.2

Architecture Body Format. The architecture body describes the structure of the combinational circuit to be diagnosed. The form of the architecture body is restricted by the subset of VHDL specified in Appendix C. In VHDL, the entity declaration of a design entity defines the input and output signals. Internal signals must be declared in the `architecture_declarative_part` of the architecture body. Each gate is represented by a single signal assignment statement in the `architecture_statement_part` of the architecture body. Construction of a signal assignment statement to represent a gate is similar to the manner in which a Boolean equation is used to denote a gate. VHDL provides operators which are identical in function to the operators used in the formation of Boolean equations. VHDL operators and their analog Boolean equation operators are given in Table 5.1.

VHDL Operator	Boolean Operator
<code>and</code>	*
<code>or</code>	+
<code>xor</code>	!
<code>not</code>	,

Table 5.1. VHDL Operators and Their Boolean Equation Analogs

An AND gate is represented two ways: with or without parentheses. Unlike the Boolean equation representation, the AND operator cannot be replaced by juxtaposition in VHDL. Examples of VHDL AND gates are:

```
output <= first_input and second_input;
output <= (first_input and second_input);
```

Multiple-input AND gates are denoted by a string of input signals and operators, e.g.,

```
output <= first_input and second_input and third_input;
```

An OR gate can also be represented with or without parentheses. Examples of VHDL OR gates representations are

```
output <= first_input or second_input;
output <= (first_input or second_input);
output <= first_input or second_input or third_input;
```

Examples of XOR gates representations are

```
output <= first_input xor second_input;
output <= (first_input xor second_input);
output <= first_input xor second_input xor third_input;
```

The VHDL NOT gate is slightly different from the Boolean-equation representation of a NOT gate. In the Boolean-equation form of a NOT gate, the ' operator is used in a postfix fashion; however, the VHDL not operator is used in a prefix manner. Two ways of representing a NOT gate using signal assignment statements are:

```
output <= not input;
output <= not (input);
```

VHDL provides NAND and NOR operators; however, the use of these operators is restricted to representing two-input gates. The reason for this restriction is that the operators are not associative [IEEE 88:7-2]. Thus, the use of these operators is excluded from the subset of VHDL used to represent circuits. Instead of using these operators, combinations of operators are used as in the case of Boolean equations. Also similar to the Boolean-equation representation, parentheses are required in the signal assignment statements that represent NAND, NOR, and XNOR gates.

Ways to represent NAND gates with signal assignment statements are:

```
output <= not (first_input and second_input);
output <= not (first_input and second_input and third_input);
```

NOR gates are represented in the following way:

```
output <= not (first_input or second_input);
output <= not (first_input or second_input or third_input);
```

Examples of XNOR gate representations are:

```
    output <= not (first_input xor second_input);
    output <= not (first_input xor second_input xor third_input);
```

Using signal assignment statements, the architecture body of the circuit of Figure 5.2 is given in Figure 5.5.

```
-- architecture body for CircuitFigure.5.2
architecture DataFlow of CircuitFigure.5.2 is
  signal D, E, F : bit;
begin
  D <= not B;
  E <= not (A and D);
  F <= not (C and D);
  Z <= not (E and F);
end DataFlow;
```

Figure 5.5. VHDL Architecture Body of the Circuit in Figure 5.2

VHDL architecture bodies representing circuits to be diagnosed by the diagnostic system may be commented normally. Additionally, time expressions are allowed in signal assignment statements. If used in VHDL simulations, these expressions are ignored by the diagnostic system.

The Equation Generation Module

In the mathematical development of the diagnostic system, the system of equations representing the circuit was used to generate the characteristic equation, $\Phi_0(\underline{x}, \underline{y}, z) = 0$. The characteristic equation incorporates the network function as well as the checkpoint information of the circuit. The equation generation module uses the intermediate format constructed by the input module to generate the characteristic equation. The characteristic equation, and lists of the circuit inputs, output, and checkpoint variables are then passed to remaining modules. A pictorial description of the equation generation module is given in Figure 5.6.

EQUATION GENERATION MODULE

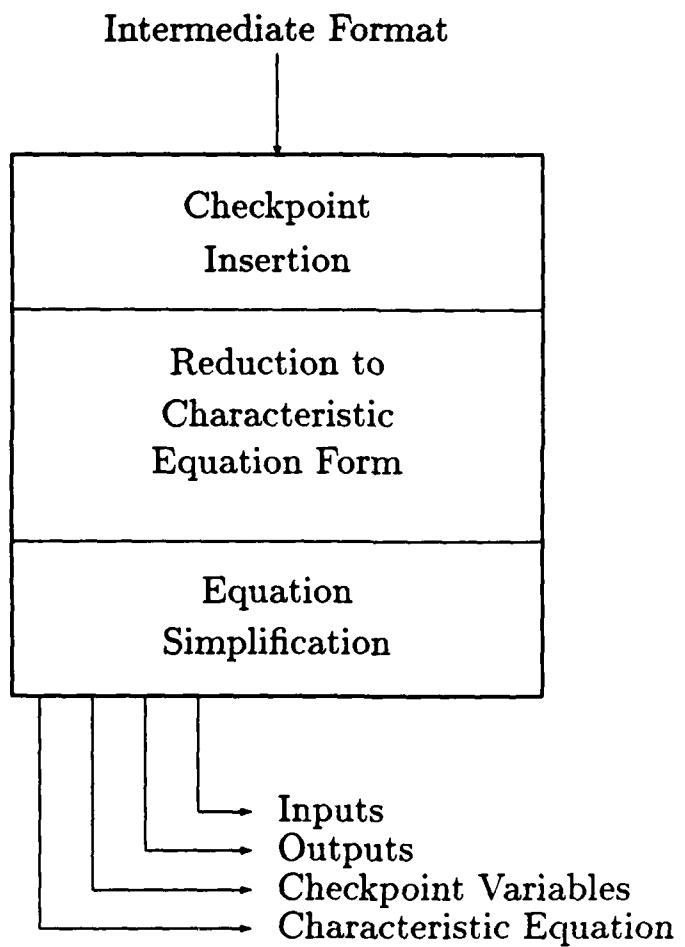


Figure 5.6. The Equation Generation Module

The specification of the intermediate format is important because it must be used in the equation generation module to determine which nodes of the circuits are inputs, the output, and the checkpoints. Two lists, one containing the circuit inputs and the other circuit output, are passed to the remaining modules of the system. The checkpoints of a circuit, fanout branches of nodes which fan out as well as primary inputs which do not fan out, must be determined in order to construct the two equations associated with checkpoint logic gates, equations (4.5) and (4.6), for each checkpoint in the circuit. Once these equations are constructed, a list must be generated containing the checkpoint variables associated with the checkpoint equations. This list is passed to the remaining modules of the diagnostic system.

The Boolean operations of reduction and elimination are used to construct the circuit characteristic equation from the gate equations and checkpoint equations. First, the checkpoint equations are constructed. Reduction is used to construct a single equation from the system of equations. Elimination is used to remove internal nodes from the single equation, since we are concerned only about the inputs, output, and checkpoint variables. Thus, the characteristic equation

$$\Phi_0(\underline{x}, \underline{y}, z) = 0 \quad (5.1)$$

is constructed, where

- \underline{x} are the input variables,
- \underline{y} are the checkpoint variables, and
- z is the output variable.

The exact order of reduction and elimination in the process of constructing the characteristic equation is implementation-dependent. After it is formed, the characteristic equation is passed to the next module in the diagnostic system.

The Tester Module

Generation of effective tests and the deduction of new information are interrelated steps in the diagnostic algorithm, because a test vector is generated after new information is deduced from the result of the previous test. The tester module incorporates the test vector generation and information deduction steps of the diagnostic algorithm developed in Chapter 4. At each iteration, the tester module specifies a test vector to be input to the potentially faulty circuit. After a test is conducted, the module accepts the result of the test, which is used to generate an updated characteristic equation. After a sufficient number of tests, the input function, $i_i(\underline{x})$, becomes equal to 1, as in equation (4.70). At this point, the testing process is concluded. The characteristic equation used to generate this input function incorporates the fault conditions as well as the function that the faulty circuit is performing. This final characteristic equation is passed to the last module of the diagnostic system for interpretation. A pictorial representation of the tester module is given in Figure 5.7.

The characteristic equation, the list of the checkpoint variables, and the circuit output list which were constructed by the equation generation module are used to generate the first test vector. The steps outlined in the Generation of Effective Test Vectors section of Chapter 4 will be implemented in the tester module. The input equation $i_i(\underline{x}) = 0$ is used to generate a test vector \underline{x} defined by $m_{j(i)}(\underline{x}) = 1$. Both the input equation and the test vector will be output to the user of the system. The input equation is output to the user in case the user would like to know the constraints that caused the diagnostic system to generate the associated test vector. Solutions of the input equation are all effective test vectors. One solution is chosen arbitrarily to be the designated test vector. The test vector that is produced will designate the signal to be applied to each primary input of the circuit during the next test. Figure 5.8 gives an example of the form of the input equation and test vector that would be output by the diagnostic system if the circuit in Figure 5.2 was being diagnosed.

TESTER MODULE

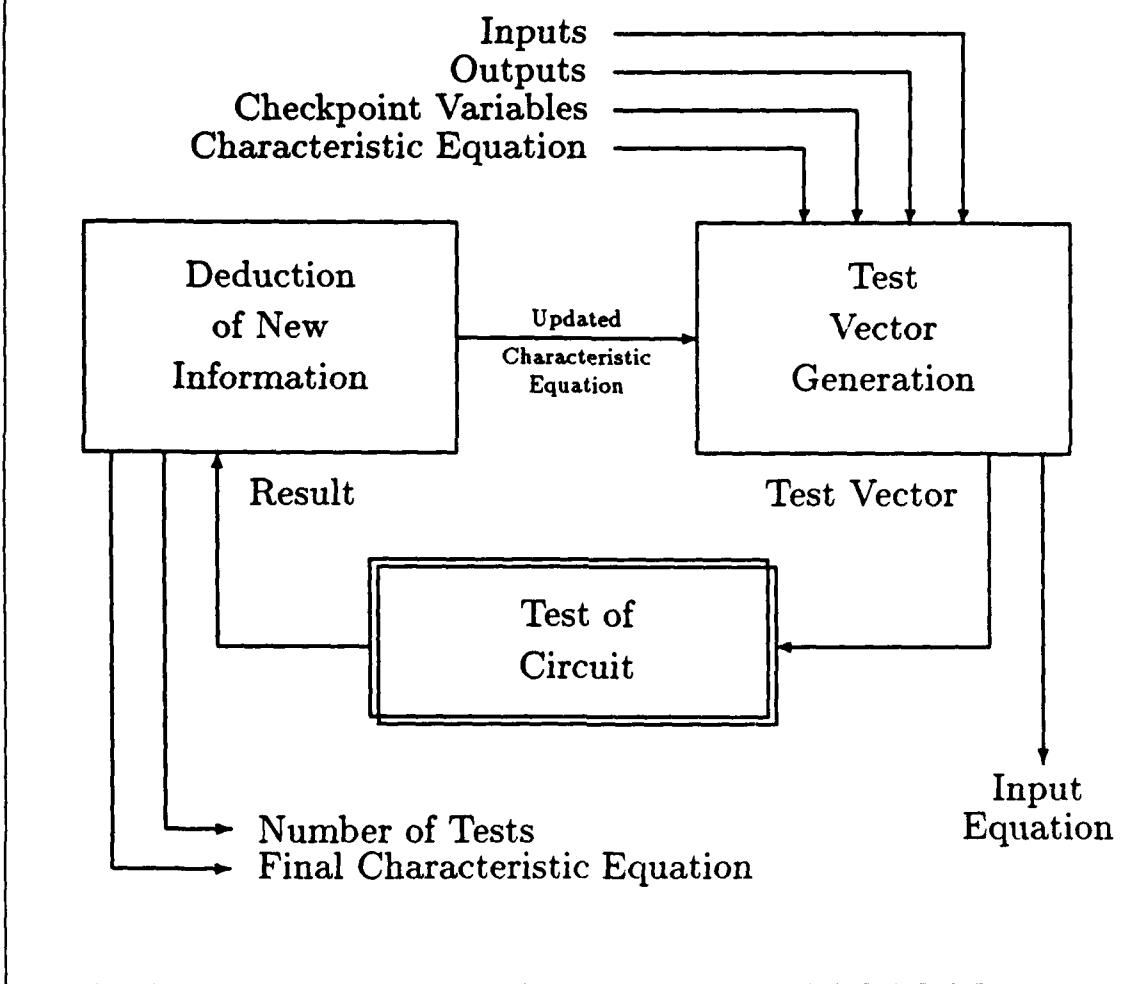


Figure 5.7. The Tester Module

The Input Equation is: $A + C = 0$
The Suggested Input is:

A = 0
B = 1
C = 0

Figure 5.8. Input Equation and Test Vector for the Circuit in Figure 5.2

The result of the test will be input to the tester module by the user. Since the circuit to be diagnosed by this diagnostic system is constrained to have a single output, the system simply asks the user whether the output of the circuit was 0 or 1 as a result of the input-output test. Following the steps outlined in the Deduction of New Information section of Chapter 4, an updated characteristic equation, $\Phi_i(\underline{x}, \underline{y}, z) = 0$, is developed using both the test vector and the result of the test in which the test vector was used. This updated equation incorporates new information regarding the state of the circuit. After the first updated characteristic equation is formed, a second input equation and test vector are generated using the updated characteristic equation, the checkpoint variables, and the circuit output. The tester module then iterates in a cycle of test vector generation, input-output test, and deduction of new information until the input function $i_{i+1}(\underline{z})$ becomes equal to 1. When this occurs, all information deducible by input-output tests has been derived. The updated characteristic equation $\Phi_i(\underline{x}, \underline{y}, z) = 0$ which was used to generate the final input function incorporates the fault conditions as well as the function the potentially faulty circuit is performing. This equation is passed to the last module of the diagnostic system.

While iterating through the test-result cycle, the number of input-output tests will be counted. This number will also be passed to the final module of the diagnostic system.

The Interpretation Module

The interpretation module uses outputs of all of the other modules to generate user-readable information regarding the fault conditions of the circuit in addition to the function that the circuit is performing. The interpretation module incorporates all steps of the diagnostic algorithm developed in Chapter 4, Interpretation of Results. Performance metrics for the diagnostic system are also developed by this module. Figure 5.9 gives a pictorial description of the interpretation module.

The first operation performed by the interpretation module is to determine the function that the potentially faulty circuit is performing. The characteristic equation output by the tester module, the checkpoint variables and the output variable are used to determine the actual function of the circuit. To give the user an idea of how the actual circuit function compares to the designed function, an equation representing the designed function of the circuit is generated using the intermediate format that was produced by the input module. Then, the actual function is compared to the designed function to determine equivalence. The actual and designed circuit functions each will be output to the user in the form of a Boolean equation. This equation will specify the circuit output as a function of the inputs. The result of the equivalence check will then be output. For the circuit of Figure 5.2, if node B were stuck-at-0, then the actual and designed circuit functions would be output to the user as shown in Figure 5.10.

Enumeration of possible fault conditions is the second operation performed by the interpretation module. Using the final characteristic equation from the tester module and the output variable, possible states of the checkpoint variables are determined. In certain cases the diagnostic system can determine the state of some checkpoints in the circuit with certainty, i.e., the input-output experiment may provide enough information to conclude that a node is definitely normal, stuck-at-0, stuck-at-1, not stuck-at-0, or not stuck-at-1. The interpretation module outputs the values of all such nodes to the user. An example of the output to the user when the state of nodes is determined with certainty is given in Figure 5.11.

INTERPRETATION MODULE

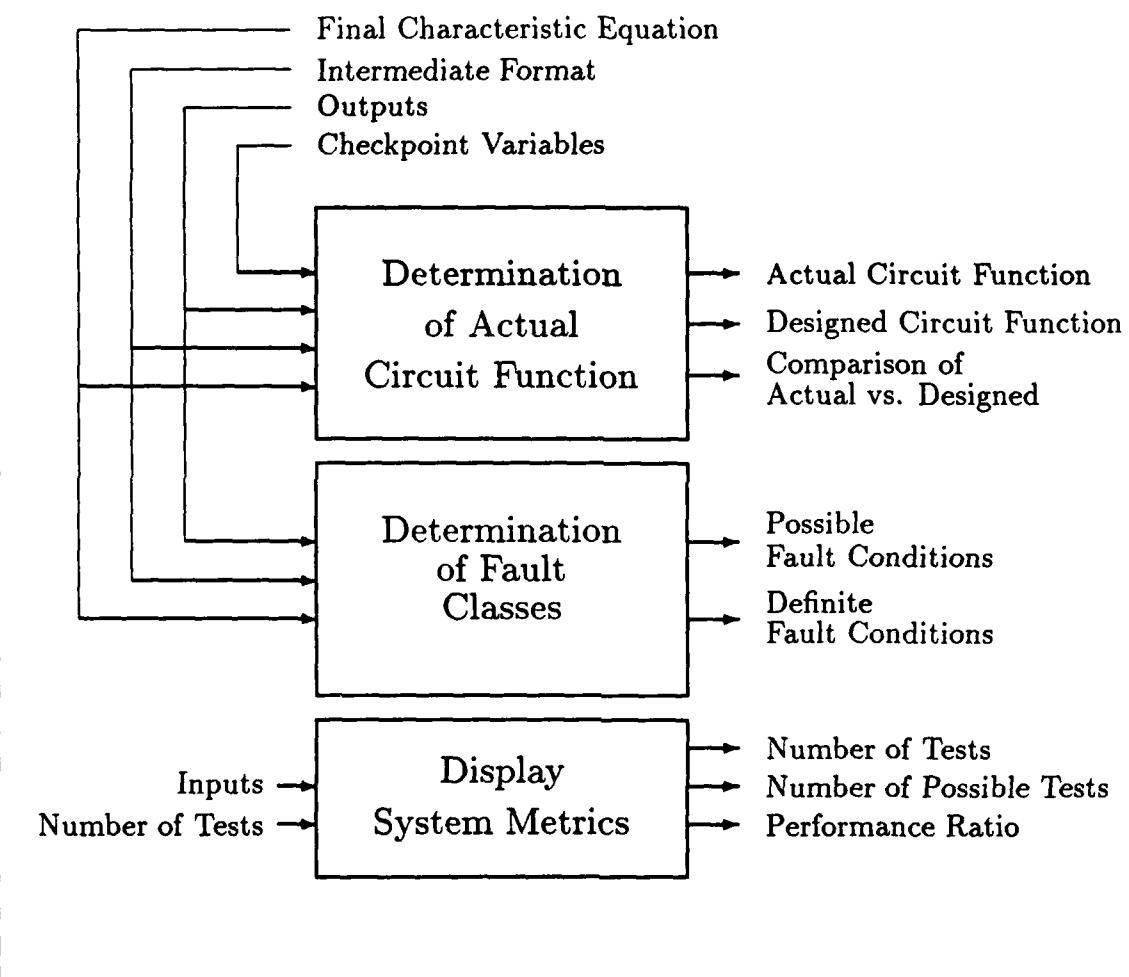


Figure 5.9. The Interpretation Module

Designed Circuit Function:

$$Z = A B' + B'C$$

Actual Circuit Function:

$$Z = A + C$$

The actual circuit IS NOT equivalent to the designed circuit.

Figure 5.10. Designed and Actual Circuit Functions for B s-a-0 in Figure 5.2

The intermediate format must be used when the fault conditions are output to the user. The intermediate format is the form to which either Boolean equations or a VHDL description is converted by the input module. The characteristic equation incorporates the inputs, output and the checkpoint variables of the circuit. However, the checkpoint variables are meaningless to the user. Thus, the interpretation module must perform a reverse process of associating checkpoint variables with specific nodes in the circuit. The intermediate format incorporates the circuit structure; hence, it is used to identify the nodes with which the checkpoint variables in the characteristic equation are associated. The condition of a specific node in the circuit is useful information for the user; therefore, the items output to the user are a node and the state of the node.

When the condition of nodes is output to the user, checkpoint nodes associated with nodes which fan out must be handled differently from checkpoint nodes associated with input nodes which do not fan out. Whereas the state of an input node which does not fan out can be clearly stated, conditions of checkpoint nodes associated with fanout branches cannot be output to the user by stating just the node label and the state of the node. All fanout branches and the fanout stem of a node which fans out have the same label. Hence, to avoid ambiguity when stating the conditions of checkpoint nodes associated with a fanout branch, the logic gate which has the fanout branch as

**** The following information is certain about the circuit. ****

Input nodes (which do not fanout) that are normal:

C

A

Input nodes (which do not fanout) that are stuck-at-0:

--none--

Input nodes (which do not fanout) that are stuck-at-1:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-0:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-1:

--none--

Fanout nodes that are normal:

--none--

Fanout nodes that are stuck-at-0:

--none--

Fanout nodes that are stuck-at-1:

--none--

Fanout nodes that are NOT stuck-at-0:

Node D of gate: $F = (C * D)'$

Node D of gate: $E = (A * D)'$

Fanout nodes that are NOT stuck-at-1:

--none--

Figure 5.11. Output of Information that is Certain About Node States

an input must be stated with the node label and the condition of the node. See Figure 5.12 for an example of a node label and associated logic gate. Figure 5.11 depicts how the checkpoint information for a fanout node is output to the user.

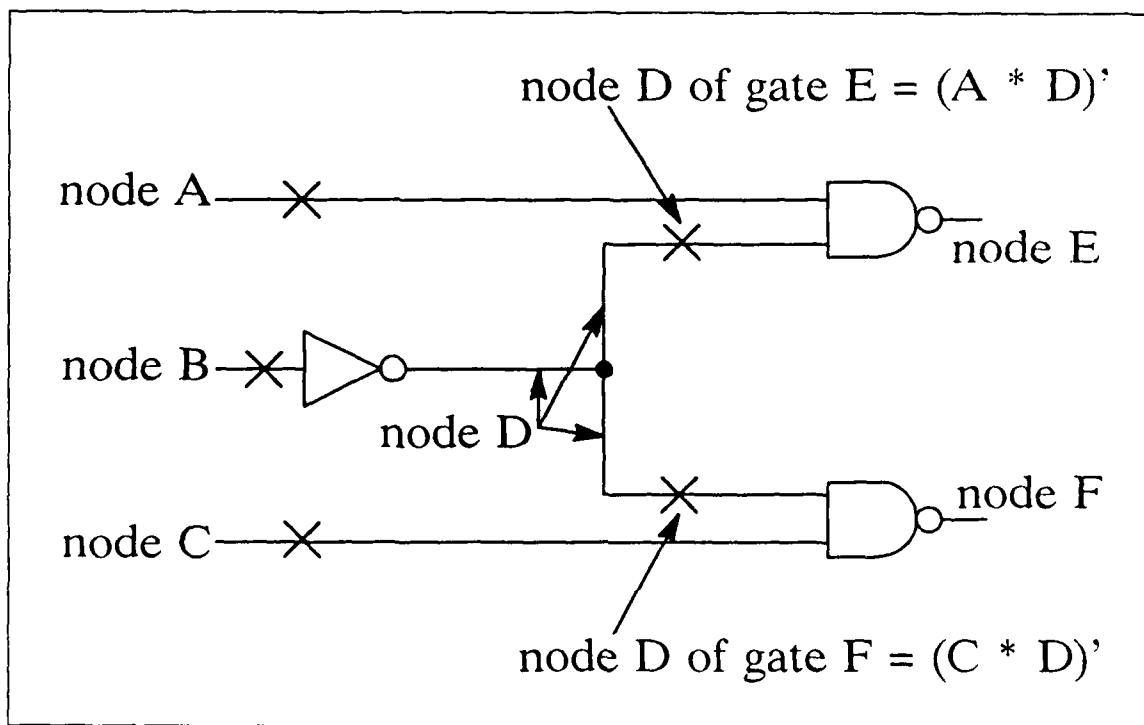


Figure 5.12. Fanout Nodes and Associated Logic Gates

After information is output to the user regarding nodes in which the state was determined with certainty (Figure 5.11), possible node conditions must be output. Checkpoint nodes in which the state is not determinable combine to form occurrences of possible faults in the circuit. In these cases, different fault conditions can cause the circuit to perform in a like fashion, i.e., the possible fault conditions are members of the same equivalence class. The interpretation module will output the possible fault conditions on a case-by-case basis. Figure 5.13 gives an example of this output. One of the possible fault conditions exists in the circuit, however, this condition cannot be distinguished by means of testing.

**** One of the following cases holds for the circuit ****

**** Case #1 ****

Input node B is stuck-at-0.

**** Case #2 ****

Input node B is not stuck-at-1.

Node D of gate: $F = (C * D)'$ is stuck-at-1.

Node D of gate: $E = (A * D)'$ is stuck-at-1.

**** Case #3 ****

Input node B is not stuck-at-0.

Node D of gate: $F = (C * D)'$ is stuck-at-1.

Node D of gate: $E = (A * D)'$ is stuck-at-1.

Figure 5.13. Output of Possible Fault Cases

The last function of the interpretation module is the determination of performance metrics. Currently, the only metric devised for the system is a comparison of the number of tests that were used in diagnosing the circuit against the number of possible input vectors that could have been applied to the circuit. As other metrics are devised, they should be incorporated into this section of the interpretation module. The number of tests used in the circuit diagnosis is output by the tester module. To determine the number of possible input vectors, the interpretation module uses the input signal list that was constructed by the equation generation module. The number of possible input vectors is 2^n , where n is the number of input signals. After the number of possible input vectors is determined, a performance ratio is devised which is a measure of the number of tests used for diagnosis of the circuit relative to the number of possible tests that could have been used. An example of performance metric output is given in Figure 5.14.

Extensions to the Architecture

The architecture outlined in this chapter was developed to implement a diagnostic system that will perform complete diagnosis of a single-output combinational circuit. Further refinement

System Performance Metrics:

The number of tests run was: 4
The number of possible tests was: 8
The performance ratio is: 0.5

Figure 5.14. Example Performance Metric Output

of the architecture would be needed to implement the extensions discussed in the last section of the previous chapter. In most cases, these extensions can be incorporated into the current architecture. Changes to the existing architecture as well as the development of additional modules are a subject for further work.

In the next chapter, the system architecture is used to describe an implementation of the diagnostic system. Details regarding the construction of each module as well as implementation-dependent data formats are described.

VI. Implementation of the Diagnostic System

In this chapter, the implementation of the diagnostic system is described. The system was developed on 80286- and 80386-based IBM-compatible computers. The version of Scheme used in this implementation is PC Scheme, version 3.0, developed by Texas Instruments.

Introduction

PC Scheme is an integrated environment which executes under the MS DOS operating system. PC Scheme provides editing and debugging tools which facilitate easy program development, although procedures may be edited outside the Scheme environment and then loaded for future evaluation. Once procedures are developed, with either PC Scheme's editor or an external editor, they are loaded into the PC Scheme environment. While being loaded, procedures are checked for proper syntax; PC Scheme prints out an error message if a procedure which is in the process of being loaded has syntactic errors. Procedures also may be defined directly at the PC Scheme prompt. [Texas 87a]

After user-defined procedures have been loaded into the environment, they may be evaluated. In addition, Scheme provides a collection of procedures called *primitive procedures* which are predefined procedures that may be evaluated at any time. Excluding minor extensions which conform to Common Lisp, primitive procedures of PC Scheme adhere to the Revised³ Report on the Algorithmic Language Scheme [Texas 87a:1-3]. The TI Scheme Language Reference Manual describes primitive procedures available in PC Scheme [Texas 87b].

When a procedure is typed at the PC Scheme prompt, PC Scheme evaluates the procedure based on a set of evaluation rules and returns the result of the evaluation. A procedure call in Scheme is a list of the form:

(<Procedure-Name> <1st Argument> <2nd Argument> ...<nth Argument>)

where <Procedure-Name>, <1st Argument>, <2nd Argument>, and <nth Argument> are distinct elements of the list. A list is always enclosed in parentheses. Except in certain cases, whenever Scheme receives a list it assumes that the first element after the left parenthesis is a procedure name and the other elements within the parentheses are arguments of the procedure [Eisen 88:20]. When the primitive procedure “+” is typed with the arguments 1 and 2 at the PC Scheme prompt, the result of adding 1 and 2, i.e., 3, is returned. An example of this process is shown in Figure 6.1. The numbers in brackets, [1] and [2], are prompts within the PC Scheme environment. The number within the brackets is incremented after each evaluation. The expression (+ 1 2) is the primitive procedure with its respective arguments. The number 3 is the result of evaluating the primitive procedure.

```
[1] (+ 1 2)
3
[2]
```

Figure 6.1. Evaluation of a Scheme Primitive Procedure

User-defined procedures are constructed using primitive procedures as well as other user-defined procedures. User-defined procedures are built using the **define** special form. A Scheme special form is one of the constructs that is evaluated differently from the manner in which procedures are evaluated. Each Scheme special form has its own unique format and associated evaluation rule [Eisen 88:20-21]. One of the functions of the **define** special form is to facilitate the construction of user-defined procedures. The format of a procedure definition using the **define** special form is given in Figure 6.2. The item <Procedure-Name> in Figure 6.2 is the new procedure name. The formal parameters of the new procedure are <1st Argument>, <2nd Argument>, ..., <nth Argument>. <procedure-body> is a sequence of statements to include procedure calls which define the operation of the new procedure.

```
(define (<Procedure-Name> <1st Argument> <2nd Argument> ...<nth Argument>)
      <procedure-body>)
```

Figure 6.2. Format for the Definition of a User-Defined Procedure

When the special form `define` is used to create a new procedure, after the `define` expression is evaluated, the procedure being defined becomes available for future use. An example of a user-defined procedure definition is given in Figure 6.3. In this example, a new procedure `double` is defined. In this procedure, the `input-number` is added to itself to return the double of the `input-number`. The expression `(+ input-number input-number)` forms the procedure body which defines the operation of the user-defined procedure `double`. Note that the primitive procedure `+` was used in constructing procedure `double`. Primitive procedures and other user-defined procedures may be used within the procedure body. A significant aspect of the Scheme language is that a procedure may be defined recursively; thus, a procedure may call itself within its own procedure body. Figure 6.4 illustrates the process of defining a new procedure and then using it within the PC Scheme environment. Note that the result of the evaluation of the `define` special form is the name of the new procedure; this informs the user that the new procedure is available for evaluation.

```
(define (double input-number)
      (+ input-number input-number))
```

Figure 6.3. An Example of a User-Defined Procedure Definition

Scheme provides mechanisms for the creation of local variables within a procedure. The values assigned to the local variables are the result of evaluating intermediate procedures within a procedure body. When a procedure is called from the system prompt, the result of the procedure's evaluation is returned to the user as shown in Figure 6.1. When a procedure is called from within a procedure body of a calling procedure, the result of its evaluation may be returned to local variables

```
[2] (define (double input-number)
(+ input-number input-number))
DOUBLE
[3] (double 2)
4
[4]
```

Figure 6.4. Defining and Using a New Procedure

within the calling procedure. Hence, local variables temporarily hold the results of intermediate evaluations within a procedure body. After local variables receive values, the local variables as well as the formal parameters of the procedure may be used as parameters when calling other procedures from within the procedure body. If a local variable, having previously received a value, is the last expression in a procedure body, then the value of the local variable is returned as the result of evaluating the original procedure.

A special case is when a procedure being called is the last statement of a calling procedure's procedure body. In this case, the result of the evaluation of the called procedure also becomes the result of the evaluation of the calling procedure. This result is returned to the location from which the calling procedure was originally called—either another procedure or the system prompt. This is demonstrated by the example given in Figure 6.4. When procedure **double** is evaluated, the result of evaluating the last statement in its procedure body, i.e., `(+ input-number input-number)`, becomes the result of evaluating procedure **double**. This result is returned to the PC Scheme prompt, because this was the location from which procedure **double** was called.

Like other programming languages, Scheme provides mechanisms for input-output. Input data may be read either from an input file or from the keyboard. Typically, data which is input is assigned to a local variable; the value of the local variable is passed to other procedures for processing. Additionally, procedures are provided to output information to the user. The results of

evaluating intermediate procedure calls, the value of local variables, and normal textual information may be output either to an output file or to the screen.

In the implementation of the diagnostic system, a main procedure was constructed from which subordinate procedures corresponding to the modules described in Chapter 5 are called. Local variables are used to pass the data, as described in Chapter 5, between the modules. An example of the main procedure definition is given in Figure 6.5.

```
(define (diagnose)
  (let* ( (intermediate-format (run-input-module))
          (phi (generate-equation intermediate-format))
          (tester-output (tester phi)) )
    (interpret intermediate-format phi tester-output)))
```

Figure 6.5. Example of the Main Procedure Definition

In Figure 6.5, the `let*` special form is the Scheme construct that facilitates the instantiation of the local variables `intermediate-format`, `phi`, and `tester-output`. Procedures `run-input-module`, `generate-equation`, `tester`, and `interpret` are user-defined procedures which establish the operation of procedure `diagnose`. When procedure `diagnose` is called from the PC Scheme prompt, the first action taken is to call procedure `run-input-module`. Procedure `run-input-module` performs all of the actions of the input module as described in Chapter 5, The Input Module (see Figure 5.1). Included in these actions are prompting the user for the type of description that will be used to represent the circuit to be diagnosed (Boolean equations or VHDL), reading the file, and converting the representation to the intermediate format. The intermediate format that is created by the input module is returned as the result of evaluating `run-input-module` and is assigned to the local variable `intermediate-format`.

The next action that occurs is a call to the `generate-equation` procedure. This procedure performs the operations associated with the equation generation module as described in Chapter 5,

The Equation Generation Module (see Figure 5.6). The local variable `intermediate-format` is used as the input parameter to the `generate-equation` procedure. The primary operation performed by procedure `generate-equation` is to use the intermediate format to form the initial characteristic function, $\Phi_0(\underline{x}, \underline{y}, z)$. Additionally, lists are created of the input variables, the checkpoint variables, and the output variable for use by other modules. However, only one object may be returned by a Scheme procedure. Hence, lists representing the characteristic equation, the input variables, the checkpoint variables, and the output variable must be combined to form a single list which is returned as the evaluation of procedure `generate-equation`.

After the local variable `phi` is assigned the result of evaluating procedure `generate-equation`, `phi` is passed as a parameter to procedure `tester`. Procedure `tester` executes the functions associated with the tester module described in Chapter 5, The Tester Module (see Figure 5.7). Initially, the list associated with the input parameter `phi` is decomposed into its requisite components. After generating a test vector, procedure `tester` outputs the test vector to the user as shown in Figure 5.8 and accepts the result of the application of the test vector to the circuit under test. Procedure `tester` derives the new information which is a result of the input-output experiment and uses this information to produce the updated characteristic equation. After determining that further information cannot be deduced from testing, `tester` combines the final characteristic equation with the number of tests that were executed to form a single list which is assigned to the local variable `tester-output` as the result of evaluating procedure `tester`.

Procedure `interpret` is the last procedure called by procedure `diagnose`. Procedure `interpret` performs the operations associated with the interpretation module as described in Chapter 5, The Interpretation Module (see Figure 5.9). The local variables `intermediate-format`, `phi`, and `tester-output` are passed as parameters to procedure `interpret`. Initially, procedure `interpret` decomposes the input parameters `phi` and `tester-output` into their component parts. Lists representing the final characteristic function, $\Phi_0(\underline{x}, \underline{y}, z)$, the checkpoint variables, the output variable,

and the intermediate format are used to determine both the actual circuit function and the designed circuit function. After these functions are determined, they are compared to determine equivalency. These results are output to the user as shown in Figure 5.10. The next operation performed by procedure **interpret** is to determine the possible fault conditions in the circuit. The state of some checkpoint nodes may be determined with certainty; using the final characteristic function, the output variable, and the intermediate format, procedure **interpret** ascertains these nodes and outputs this information to the user (Figure 5.11). Because the condition of other checkpoint nodes is uncertain; procedure **interpret** determines the possible conditions of the remaining checkpoint nodes and outputs these conditions as shown in Figure 5.13.

The final operation executed by procedure **interpret** is to use the list of the circuit inputs and the number of tests that were performed to develop performance metrics of the diagnostic system. This information is displayed for the user as depicted in Figure 5.14. Once procedure **interpret** is evaluated, this result also becomes the result of evaluating procedure **diagnose**. Since procedure **diagnose** was called from the PC Scheme environment, after **diagnose** is evaluated, the user is returned to the PC Scheme prompt.

Procedures **run-input-module**, **generate-equation**, **tester**, and **interpret** are all constructed in a manner similar to procedure **diagnose**. Appendix D, Diagnostic System Code, gives the fully-commented source code for all procedures of the diagnostic system. Each procedure includes comments which describe its operation. The system is decomposed into separate files which parallel the system architecture described in Chapter 5. Each file includes a header which describes in general terms the operations performed by the procedures in the file. For further information regarding the Scheme language see either *Programming in Scheme* by Michael Eisenberg [Eisen 88] or *Structure and Interpretation of Computer Programs* by Abelson and Sussman [Abels 85].

In the remainder of the chapter, the approach taken is to describe the transformation of the data as it flows through the system. Additionally, design decisions important to the implementation

of each module are discussed. Initially, a complete diagnostic session will be shown; this session will be a point of reference for the remaining sections of the chapter.

A Diagnostic Session

After entering the PC Scheme environment, the files containing the procedures that implement the diagnostic system must be loaded. This is performed using the PC Scheme `load` procedure [Texas 87b:7-100]. Alternatively, it is possible to pre-load these files, i.e., when the PC Scheme environment is entered from the MS DOS prompt, the files are loaded automatically. These options are described in the PC Scheme User's Manual [Texas 87a].

Once the diagnostic system is loaded, diagnosis begins by typing (`diagnose`) at the PC Scheme prompt as shown in Figure 6.6. Figures 6.6, 6.7, 6.8, and 6.9 demonstrate a complete diagnostic session. The circuit under diagnosis is from Figure 5.2. The Boolean equation description for this circuit is given by Figure 5.3; the VHDL representation is shown in Figure 5.5. The responses for each of the suggested test vectors reflect the response of the circuit under the condition that the input node `b` is stuck-at-0. After the results of the diagnostic session are output to the user, control is returned to the PC Scheme prompt.

All information regarding the results of the diagnostic experiment is output directly to the screen rather than to a file. The rationale for this is that PC Scheme provides a transcript mechanism which can be used to record in a file everything that was input and output to the console (keyboard/screen) [Texas 87b:7-186]. The transcript mechanism was used to record the session shown in Figures 6.6, 6.7, 6.8, and 6.9.

The Input Module

After the user selects the form of the circuit representation, i.e., Boolean equations or VHDL description, the file which holds the representation must be read. Regardless of the representation,

[3] (diagnose)

Enter the type of input file for your circuit description.
Your choice are:

1. VHDL Dataflow Description
2. Boolean Equation Description

Enter the number of your choice --> 1

**** Enter the filename of your input file ****

Enter the input filename --> circuit1.vhd

Processing....

The Input Equation is: $O = 0$

The Suggested Input is:

A = 1
B = 1
C = 1

If the output was 0, type 0 and <rtn>, else type 1 and <rtn>.
Enter the Resulting Output (0 or 1) --> 1

Processing....

The Input Equation is: $A C = 0$

The Suggested Input is:

A = 0
B = 1
C = 1

Enter the Resulting Output (0 or 1) --> 1

Processing....

The Input Equation is: $C = 0$

The Suggested Input is:

A = 1
B = 1
C = 0

Enter the Resulting Output (0 or 1) --> 1

Figure 6.6. A Diagnostic Session (Screen 1)

Processing....

The Input Equation is: $A + C = 0$
The Suggested Input is:

$A = 0$
 $B = 1$
 $C = 0$

Enter the Resulting Output (0 or 1) --> 0

Processing....

The Input Equation is: $1 = 0$
New information cannot be obtained.

***** Results *****

The function that the circuit was designed to perform is:

$Z = B'C + A'B'$

The function that the circuit is performing is:

$Z = A + C$

The actual circuit IS NOT equivalent to the designed circuit.

**** The following information is certain about the circuit ****

Input nodes (which do not fanout) that are normal:

C
A

Input nodes (which do not fanout) that are stuck-at-0:

--none--

Input nodes (which do not fanout) that are stuck-at-1:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-0:

--none--

Figure 6.7. A Diagnostic Session (Screen 2)

```
Input nodes (which do not fanout) that are NOT stuck-at-1:  
--none--  
  
Fanout nodes that are normal:  
--none--  
  
Fanout nodes that are stuck-at-0:  
--none--  
  
Fanout nodes that are stuck-at-1:  
--none--  
  
Fanout nodes that are NOT stuck-at-0:  
Node D of gate: F = (C * D)'  
Node D of gate: E = (A * D)'  
  
Fanout nodes that are NOT stuck-at-1:  
--none--  
  
**** One of the following cases holds for the circuit ****  
**** Case #1 ****  
Input node B is stuck-at-0.  
Press <return> to continue.  
**** Case #2 ****  
Input node B is not stuck-at-1.  
Node D of gate: F = (C * D)' is stuck-at-1.  
Node D of gate: E = (A * D)' is stuck-at-1.  
Press <return> to continue.  
**** Case #3 ****  
Input node B is not stuck-at-0.  
Node D of gate: F = (C * D)' is stuck-at-1.  
Node D of gate: E = (A * D)' is stuck-at-1.
```

Figure 6.8. A Diagnostic Session (Screen 3)

```
Press <return> to continue.  
  
System Performance Metrics:  
  
The number of tests run was: 4  
The number of possible tests was: 8  
  
The performance ratio is: 0.5  
  
(  
[4]
```

Figure 6.9. A Diagnostic Session (Screen 4)

after all of the characters of the input file are read, they are placed into a list. Because the contents of the input file are placed into a list, the maximum length of an input file is constrained. In PC Scheme, the number of elements of a list is constrained only by the amount of heap space available in the system. However, the `length` primitive procedure used to determine the number of elements in a list assumes that a list may have no more than 16,383 elements [Texas 87a:10-2]. The `length` procedure is used in this implementation of the diagnostic system. Hence, the maximum number of characters allowed in an input file is 16,383. After the list of characters is formed, the different representations are handled in slightly different fashions.

Boolean Equation Input. For the input file of Boolean equations given by Figure 5.3, the list of characters is shown in Figure 6.10. ASCII characters in PC Scheme are prefaced by the `#\` notation [Texas 87b:4-5].

```
(#\d #\SPACE #\= #\SPACE #\b #\' #\NEWLINE  
#\e #\SPACE #\= #\SPACE #\(\ #\a #\SPACE #\d #\)\ #\' #\NEWLINE  
#\f #\SPACE #\= #\SPACE #\(\ #\c #\SPACE #\d #\)\ #\' #\NEWLINE  
#\z #\SPACE #\= #\SPACE #\(\ #\e #\SPACE #\f #\)\ #\' #\NEWLINE )
```

Figure 6.10. List of Characters (Boolean Equation Representation)

The list of characters is processed to form a list of symbols, or tokens, which includes the node labels, Boolean operators, parentheses, and symbols representing the end of a line. In the process of creating the list of symbols, all characters representing the node labels are converted to upper case. The reason for this is that manipulation of lower case characters is inconvenient in PC Scheme; hence, input to the diagnostic system is not case-sensitive. All #\SPACE characters are ignored, because they do not provide any information. Additionally, #\NEWLINE characters are replaced by .| symbols to denote the end of a line, because #\NEWLINE characters cannot be converted to a symbol which is easy to manipulate.¹ The list of symbols formed using the characters of Figure 6.10 is shown in Figure 6.11.

```
(D = B |'|. |. | E = |(| A D |)| |'|. |. | F = |(| C  
D |)| |'|. |. | Z = |(| E F |)| |'|. |.)
```

Figure 6.11. List of Symbols (Boolean Equation Representation)

After the list of symbols is formed, a revised list of symbols is formed in which node symbols which are adjacent have a * symbol inserted between them. Adjacent node symbols represent the AND operation.² Although the AND operation may be denoted either with juxtaposition or the * operator, an operator is necessary when devising the intermediate format. Thus, juxtapositions are replaced by the * symbol. The revised list of symbols is shown in Figure 6.12.

```
(D = B |'|. |. | E = |(| A * D |)| |'|. |. | F = |(| C  
* D |)| |'|. |. | Z = |(| E * F |)| |'|. |.)
```

Figure 6.12. Revised List of Symbols (Boolean Equation Representation)

The revised list of symbols is used to construct the intermediate format which represents the circuit under test. It was intended that a true parser be implemented to perform this task. However,

¹In PC Scheme, certain punctuation symbols are enclosed in bars to distinguish them from like symbols used in Scheme expressions.

²See Chapter 5, The Input Module, Boolean Equation Format.

it was more efficient to perform a direct translation from the revised list to the intermediate format. Each equation of the original input file is demarcated in the revised list of symbols by a `| . |` symbol. Hence, sets of symbols corresponding to an equation are processed one at a time. For each equation, a prefix-form list is constructed. The intermediate format for a circuit consists of a set of prefix-form lists, one for each equation (gate) of the circuit. A prefix-form was chosen, because this form is very easy to convert to the equation representation which will be described in later sections. While forming the intermediate format, `| . |` operators are replaced by the `NOT` symbol. The intermediate format developed from the list of symbols of Figure 6.12 is given in Figure 6.13. Construction of a true parser to perform the conversion from the list of symbols to the intermediate format is a subject for further work.

```
((EQ D (NOT B))
 (EQ E (NOT (* A D)))
 (EQ F (NOT (* C D)))
 (EQ Z (NOT (* E F))))
```

Figure 6.13. Intermediate Format

The last operation performed by `run-input-module` is to add a “`--`” suffix is added to symbols which represent circuit nodes. The purpose of this suffix will be discussed in later sections. The final intermediate format is depicted in Figure 6.14.

```
((EQ D-- (NOT B--))
 (EQ E-- (NOT (* A-- D--)))
 (EQ F-- (NOT (* C-- D--)))
 (EQ Z-- (NOT (* E-- F--))))
```

Figure 6.14. Intermediate Format (Suffixes Added)

VHDL Input. If the input file is the VHDL description shown in Figure 5.5, the list of characters is given by Figure 6.15.

```
(#\NEWLINE #\-- #\SPACE #\a #\r #\c #\h #\i #\t #\e #\c  
#\t #\u #\r #\e #\SPACE #\d #\e #\c #\l #\a #\r #\a #\t #\i  
#\o #\n #\SPACE #\f #\o #\r #\SPACE #\C #\i #\r #\c #\u #\i  
#\t #\_ #\l #\NEWLINE #\NEWLINE #\a #\r #\c #\h #\i #\t #\e  
#\c #\t #\u #\r #\e #\SPACE #\D #\a #\t #\a #\F #\i #\o #\w  
#\SPACE #\o #\f #\SPACE #\C #\i #\r #\c #\u #\i #\t #\_ #\l  
#\SPACE #\i #\s #\NEWLINE #\SPACE #\SPACE #\SPACE #\s #\i  
#\g #\n #\a #\l #\SPACE #\D #\, #\SPACE #\E #\, #\SPACE #\F  
#\SPACE #\: #\SPACE #\b #\i #\t #\; #\NEWLINE #\NEWLINE #\b  
#\e #\g #\i #\n #\NEWLINE #\NEWLINE #\SPACE #\SPACE #\SPACE  
#\D #\SPACE #\< #\= #\SPACE #\n #\o #\t #\SPACE #\B #\;  
#\NEWLINE #\SPACE #\SPACE #\SPACE #\E #\SPACE #\< #\=  
#\SPACE #\n #\o #\t #\SPACE #\# #\A #\SPACE #\a #\n #\d  
#\SPACE #\D #\) #\; #\NEWLINE #\SPACE #\SPACE #\SPACE #\F  
#\SPACE #\< #\= #\SPACE #\n #\o #\t #\SPACE #\# #\C #\SPACE  
#\a #\n #\d #\SPACE #\D #\) #\; #\NEWLINE #\SPACE #\SPACE  
#\SPACE #\Z #\SPACE #\< #\= #\SPACE #\n #\o #\t #\SPACE #\#  
#\E #\SPACE #\a #\n #\d #\SPACE #\F #\) #\; #\NEWLINE  
#\NEWLINE #\e #\n #\d #\SPACE #\D #\a #\t #\a #\F #\l #\o  
#\w #\; #\NEWLINE #\NEWLINE)
```

Figure 6.15. List of Characters (VHDL Representation)

Similar to the way in which the Boolean equation representation is processed, the list of characters from the VHDL input file is used to form a list of symbols. The symbols include identifiers, #\NEWLINE characters, VHDL operators, and other symbols allowable in VHDL expressions. All symbols are converted to uppercase; this is acceptable because VHDL is not case-sensitive [IEEE 88:13-4]. All #\SPACE characters are ignored, because they do not provide any information. #\NEWLINE characters are not immediately removed, because they must be used when extracting comments from the input description. The list of symbols formed using the characters of Figure 6.15 is shown in Figure 6.16.

After the initial list of symbols is created, a new list is formed in which symbols representing comments are removed. In VHDL, a comment begins with the symbol --. A comment continues

```
(#\newline -- ARCHITECTURE DECLARATION FOR CIRCUIT_1 #\newline
#\newline ARCHITECTURE DATAFLOW OF CIRCUIT_1 IS #\newline
SIGNAL D ,| E ,| F |:| BIT |;| #\newline #\newline BEGIN
#\newline #\newline D <= NOT B |;| #\newline E <= NOT |(| A AND D |)| |;|
#\newline F <= NOT |(| C AND D |)| |;|
#\newline Z <= NOT |(| E AND F |)| |;| #\newline #\newline
END DATAFLOW |;| #\newline #\newline)
```

Figure 6.16. List of Symbols (VHDL Representation)

until the end of a line. Hence, to form a revised list of symbols, all symbols between a -- symbol and the first occurrence of #\newline symbol are removed. Then, -- and #\newline symbols are removed. The revised symbol list is shown in Figure 6.17.

```
(ARCHITECTURE DATAFLOW OF CIRCUIT_1 IS SIGNAL D ,| E ,| F |:|
BIT |;| BEGIN D <= NOT B |;| E <= NOT |(| A AND D |)| |;| F
<= NOT |(| C AND D |)| |;| Z <= NOT |(| E AND F |)| |;| END
DATAFLOW |;|)
```

Figure 6.17. First Revised List of Symbols (VHDL Representation)

In the VHDL description of the circuit under diagnosis, the only information that is required to form the equations used in the diagnostic system is located between the BEGIN and END symbols. Therefore, all symbols up to and including the BEGIN symbol may be removed from the list of symbols shown in Figure 6.17. Likewise, all symbols after and including the END symbol may be removed. If any signal assignment statements include a time expression, these expressions also would be removed at this time. The resulting list of symbols is given in Figure 6.18.

```
(D <= NOT B |;| E <= NOT |(| A AND D |)| |;| F <=
NOT |(| C AND D |)| |;| Z <= NOT |(| E AND F |)| |;|)
```

Figure 6.18. Second Revised List of Symbols (VHDL Representation)

The list of symbols of Figure 6.18 is used to construct the intermediate format which represents the circuit under test. A direct translation process is performed. Each VHDL signal assignment statement is used to construct a prefix-form list corresponding to a gate in the circuit. Signal assignment statements are demarcated by `|;|` symbols. Sets of symbols corresponding to a signal assignment statement are processed one at a time. The intermediate format for a circuit consists of a set of prefix-form lists, one for each gate of the circuit. When forming the intermediate format, VHDL `AND` operators are replaced by `*` operators; `OR` operators are replaced by `+` operators; and `XOR` operators are replaced by `!` operators. The resulting intermediate format is given by Figure 6.13. Suffixes are added to node symbols resulting in the list in Figure 6.14. Boolean equations and VHDL descriptions representing the same circuit are transformed to the same intermediate format.

VHDL Description Validation. To insure that the subset of VHDL defined for use in the diagnostic system was acceptable, circuits diagnosed by the system were described using VHDL descriptions as illustrated in Figures 5.4 and 5.5. After all descriptions were constructed, they were analyzed and simulated using a VHDL 1076-based analyzer and simulator. In all cases, circuit descriptions analyzed and simulated properly.

The Equation Generation Module

As depicted in Figure 5.6, the intermediate format must be used to produce the initial characteristic equation, and lists of the circuit inputs, the checkpoint variables, and the circuit outputs.

Generation of Input and Output Lists. Formation of lists of the circuit inputs and circuit outputs is done using the intermediate format shown in Figure 6.14. Each prefix-form list of the intermediate format represents a gate in the circuit. Examining the structure of a prefix-form list, the first element of the list is a `EQ` symbol which designates an equality. The second element of the list is a single node symbol which is the output of the gate; each logic gate has exactly one

output. The last element is a list of nested operators and symbols representing the inputs to and the function of the respective gate. For example, given the prefix-form list

```
(EQ E-- (NOT (* A-- D--))) ,
```

the gate represented is a **NAND** with an output of **E--** and inputs of **A--** and **D--**. Given a set of prefix-form lists which represent a circuit, one can determine the inputs and outputs of the circuit by examining the inputs and output of the gates represented by the prefix-form lists. Inputs to the circuit are those nodes which are inputs to gates, but not outputs. Conversely, outputs from the circuit are nodes which are outputs from gates, but never an input. For the format given by Figure 6.14, a list of the inputs to the circuit is (**A-- B-- C--**); a list of the output of the circuit is (**Z--**). Using the same idea, fanout nodes and nodes internal to a circuit may also be identified. A fanout node is a node which is an input to more than one gate; node **D--** is a fanout node. Internal nodes are nodes which are both outputs from and inputs to gates in the circuit; a list of internal nodes is (**D-- E-- F--**).

The Use of the $f = 0$ Form. To form the list of the checkpoint variables, the initial characteristic equation must first be constructed. Given an initial system of equations, Boolean reduction may be used to form a single equation which is equivalent to the system of equations.³ There exist two alternative methods for forming this single equation, the $f(\underline{x}) = 0$ form of equation B.48 and the $p(\underline{x}) = 1$ of equation B.52. Rudeanu stated that the choice of one form over the other is dependent on the application [Rudea 74:52]. In Chapter 4, derivation of the characteristic equation was performed using the $f(\underline{x}) = 0$ form. This form was chosen because for the initial form of the system of Boolean equations given by the intermediate format, the time and space computational complexity for its derivation was significantly smaller than construction of the $p(\underline{x}) = 1$ form. Results of processing times to form a single equation in both $f(\underline{x}) = 0$ and $p(\underline{x}) = 1$ forms

³See Appendix B, Reduction.

for two different circuits is given in Table 6.1.⁴ The time listed as >20 minutes is the processing time before exhaustion of memory. For any system of equations, the $f(\underline{x}) = 0$ and the $p(\underline{x}) = 1$ form of the system of equations had the same number of terms. However, terms of the $p(\underline{x}) = 1$ form had many more literals than terms of the equivalent $f(\underline{x}) = 0$ form. Hence, the $f(\underline{x}) = 0$ form was used in the implementation of the diagnostic system.

Circuit	Time to Form $f = 0$	Time to Form $p = 1$
Figure 3.1	6 seconds	2:35 minutes
Figure 5.2	1:25 minutes	> 20 minutes

Table 6.1. Processing Times to Form Single Equation

The Choice of Data Structure. The intermediate format must be used to form a single equation in sum-of-products form. The data structure that represents the sum-of-products form must be one which facilitates the operations outlined in Chapter 4, Mathematical Development of the Diagnostic System. Brown [Brown 88b] and Fausett [Fause 86] have each studied in depth the issue of Scheme data structure representations of Boolean equations. The data structure used in this implementation is taken from their research.

A sum-of-products formula is a disjunction of terms. Each term of the formula is composed of literals in complemented and uncomplemented form. A way of representing a formula is by forming a list in which each term of the formula is a sublist. Literals of a term are elements of the respective sublist; if a literal is complemented, it is enclosed in a sublist—otherwise it is not. For example, given the formula $ab'c + a'bd + ab'e'$, the data structure that represents the formula is:

$$((a \ (b) \ c) \ ((a) \ b \ d) \ (a \ (b) \ (e))).$$

To form an equation, a formula is set equal to 0 or 1. Since the $f = 0$ form of a Boolean equation is used, all formulas in this implementation are equal to 0. However, this fact does not have to be

⁴The computer used to produce these results was a 20MHz 80386-based computer.

shown explicitly. Hence, when equations are represented by the data structure shown above, the fact that the respective formula is equal to 0 will be implied. Another aspect of this data structure is that a zero is represented by the null list (); a one is depicted by the (()) list.

Formation of the Characteristic Equation. A sum-of-products formula is formed using the intermediate format. As discussed in Chapter 4, for each checkpoint in the circuit there exists a checkpoint equation which must be integrated into the system of equations representing the circuit to form the characteristic equation. Checkpoints are the fanout branches of nodes which fan out and input nodes which do not fan out. Because all fanout branches and the fanout stem of a node which fans out have the same node label, a method was devised to distinguish between particular fanout branches. Otherwise, when the sum-of-products formula is constructed, the distinguishability of each fanout branch would be lost. Hence, a unique identifier is substituted for the labels associated with a fanout branch. This action must be performed while the circuit representation is in the intermediate format, because the structure of the circuit is implicit in this form.

Each fanout branch for a fanout node is assigned a number from 0 to 9. It is assumed that a fanout node will have no greater than ten fanout branches. This number is substituted in place of the first - of the -- suffix of labels associated with fanout branches. The intermediate format formed by making the fanout branches unique is given in Figure 6.19. The addition of suffixes to node labels was performed in part to handle the problem of replacing fanout branches labels with distinct identifiers. A suffix simply could not be concatenated to a label; depending on how the user labeled the remaining nodes in the circuit, concatenation of a suffix could cause conflicts in label names. Hence, a -- suffix was added to all labels and then modified as required.

After fanout branches are identified with unique labels, the intermediate format is used to develop a sum-of-products representation of the circuit. The sum-of-products representation for the intermediate format of Figure 6.19 is given in Figure 6.20.

```
((EQ D-- (NOT B--))
 (EQ E-- (NOT (* A-- D0-)))
 (EQ F-- (NOT (* C-- D1-)))
 (EQ Z-- (NOT (* E-- F-))))
```

Figure 6.19. Intermediate Format (Unique Fanout Branch Labels)

```
((D-- B--) ((D--) (B--)) (E-- A-- D0-) ((E--) A-- (D0- )))
 ((E--) (A--)) (F-- C-- D1-) ((F--) C-- (D1-)) ((F--) (C--))
 (Z-- E-- F--) ((Z--) E-- (F-)) ((Z--) (E--)))
```

Figure 6.20. Sum-of-Products Representation

Just as the labels associated with fanout branches are modified, the labels of input nodes which do not fan out are similarly changed. In these cases, the -- suffix is replaced by a X- suffix. This process may be performed after the sum-of-products representation is formed because all labels associated with nodes of this case are changed in a like manner, i.e. there is not a distinguishability problem as in the case of fanout branches. The sum-of-products representation after the revision of the input node labels is given in Figure 6.21.

```
((D-- BX-) ((D--) (BX-)) (E-- AX- D0-) ((E--) AX- (D0- )))
 ((E--) (AX-)) (F-- CX- D1-) ((F--) CX- (D1-)) ((F--) (CX-))
 (Z-- E-- F--) ((Z--) E-- (F-)) ((Z--) (E--)))
```

Figure 6.21. Sum-of-Products Representation (Revised Input Node Labels)

Once a sum-of-products representation for the circuit is formed as shown in Figure 6.21, checkpoint equations may be added. For each checkpoint, a checkpoint equation is formed as given by equation (4.5). Reduction is used to place checkpoint equation into $f = 0$ form. By equation (B.32), the checkpoint equation may be added to the sum-of-products form of Figure 6.21. Elimination is used to remove variables from the resulting equation which are not required for further processing. The process of equation reduction, addition, and elimination is conducted in an

iterative fashion until the checkpoint equations for each checkpoint in the circuit have been added to the sum-of-products form given in Figure 6.21.

For example, for the input node represented by the label **BX-**, the checkpoint equation (4.5) for the node is given by

$$\text{BX-} = \text{BX1} + \text{B-- BX0}',$$

where **BX0** and **BX1** are the checkpoint variables, **B--** is the input to the checkpoint node, and **BX-** is the output from the checkpoint node. The second character of the label suffix is used to facilitate the formation of the circuit checkpoint variables. Using reduction, the checkpoint equation can be transformed into sum-of-products form

$$((\text{BX-} (\text{B--}) (\text{BX1})) (\text{BX- BX0} (\text{BX1})) ((\text{BX-}) \text{BX1}) ((\text{BX-}) \text{B--} (\text{BX0}))).$$

This representation is added to the form given in Figure 6.21 resulting in the sum-of-products form in Figure 6.22. After this equation is formed, the variable **BX-** is not required for further processing. Following the idea illustrated in Figure 4.5, **BX-** can be viewed as the output of a checkpoint logic gate—an internal node of the circuit. The only variables required in the characteristic equation are the circuit inputs, the circuit output, and the checkpoint variables. Hence, by the property demonstrated by equation (B.62), elimination is used to eliminate the variable **BX-** from the equation in Figure 6.22. The resulting form is given in Figure 6.23.

$$((\text{D-- BX-}) ((\text{D--}) (\text{BX-})) (\text{E-- AX- D0-}) ((\text{E--}) \text{AX-} (\text{D0-}))) \\ ((\text{E--}) (\text{AX-})) (\text{F-- CX- D1-}) ((\text{F--}) \text{CX-} (\text{D1-})) ((\text{F--}) (\text{CX-})) \\ (\text{Z-- E-- F--}) ((\text{Z--}) \text{E--} (\text{F--})) ((\text{Z--}) (\text{E--})) \\ (\text{BX-} (\text{B--}) (\text{BX1})) (\text{BX- BX0} (\text{BX1})) ((\text{BX-}) \text{BX1}) ((\text{BX-}) \text{B--} (\text{BX0})))$$

Figure 6.22. Sum-of-Products Representation (With Checkpoint Equation Added)

```

(((D--) (B--) (BX1)) ((D--) BX0 (BX1)) (D-- BX1)
(D-- B-- (BX0)) (E-- AX- D0-) ((E--) (AX-))
((E--) (D0-)) (F-- CX- D1-) ((F--) (CX-)) ((F--) (D1- ))
(Z-- E-- F--) ((Z--) (E--)) ((Z--) (F--)))

```

Figure 6.23. Sum-of-Products Representation (After Elimination of BX-)

In the same manner as BX-, the checkpoint equations for checkpoints AX-, CX-, D0-, and D1- are introduced. The variables AX-, CX-, D0-, and D1- are eliminated from the equation leaving the equation in sum-of-products form shown by Figure 6.24.

```

(((F--) (C--) (CX1)) ((F--) CX0 (CX1)) (F-- CX1 D-- (D10))
(F-- C-- (CX0) D-- (D10)) (F-- CX1 D11) (F-- C-- (CX0) D11)
((D--) (B--) (BX1)) ((D--) BX0 (BX1)) (D-- BX1) (D-- B-- (BX0))
((Z--) (E--)) ((F--) (Z--)) (Z-- E-- F--) ((E--) (D--) (D01))
((E--) D00 (D01)) ((F--) D10 (D11)) ((F--) (D--) (D11))
(E-- A-- (AX0) D01) (E-- AX1 D01) (E-- A-- (AX0) D-- (D00))
(E-- AX1 D-- (D00)) ((E--) AX0 (AX1)) ((E--) (A--) (AX1)))

```

Figure 6.24. Sum-of-Products Representation (After Insertion of Checkpoint Equations)

Once the sum-of-products form shown given by Figure 6.24 is derived, variables associated with internal nodes of the circuit may be eliminated. The resulting sum-of-products form, given by Figure 6.25, consists only of input variables, the output variable, and checkpoint variables. The sum-of-products form is depicted in Blake canonical form.

Before the formation of the characteristic equation is complete, constraints associated with each checkpoint must be added to the sum-of-products form. A constraint equation for a checkpoint is given by equation (4.6). The constraint equation associated with the checkpoint examined above is given by

$$BX0 \cdot BX1 = 0.$$

```

(((B--) (BX1) C-- (CX0) (Z--) (D10)) ((B--) (BX1) CX1 (Z--) (D10))
(B-- (BX0) (D11) (D01) Z--) (B-- (BX0) (D11) Z-- AX0 (AX1))
(B-- (BX0) (D11) Z-- (A--) (AX1)) ((CX1) (C--) B-- (BX0) (D01) Z--)
((CX1) CX0 B-- (BX0) (D01) Z--) (BX0 (BX1) C-- (CX0) (Z--) (D10))
(BX0 (BX1) CX1 (Z--) (D10)) (BX1 (D11) (D01) Z--)
((D11) D10 Z-- D00 (D01)) (BX1 (D11) Z-- AX0 (AX1))
((D11) D10 Z-- AX0 (AX1)) (BX1 (D11) Z-- (A--) (AX1))
((D11) D10 Z-- (A--) (AX1)) ((CX1) (C--) BX1 (D01) Z--)
((C--) (CX1) Z-- D00 (D01)) ((C--) (CX1) Z-- AX0 (AX1))
((C--) (CX1) Z-- (A--) (AX1)) (C-- (CX0) (Z--) D11)
((CX1) CX0 BX1 (D01) Z--) (CX0 (CX1) Z-- D00 (D01))
(CX0 (CX1) Z-- (A--) (AX1)) ((B--) (BX1) (Z--) A-- (AX0) (D00))
((B--) (BX1) (Z--) AX1 (D00)) (BX0 (BX1) (Z--) A-- (AX0) (D00))
(BX0 (BX1) (Z--) AX1 (D00)) ((Z--) A-- (AX0) D01) ((Z--) AX1 D01)
(CX0 (CX1) Z-- AX0 (AX1)) (CX1 (Z--) D11) .

```

Figure 6.25. Sum-of-Products Representation (After Elimination of Internal Node Variables)

The complete constraint equation in sum-of-products form for all of the checkpoints in the example is

$$((D10 \ D11) \ (CX0 \ CX1) \ (BX0 \ BX1) \ (D00 \ D01) \ (\mathbf{AX0} \ \mathbf{AX1})).$$

This constraint is added to the form of Figure 6.25 to form the initial characteristic equation. Simplification is performed resulting in the equation in Figure 6.26. Additionally, the constraint equation is used to construct the list of checkpoint variables that is required by other modules in the system. A single list which includes the initial characteristic equation, lists of the input variables, the checkpoint variables, and the circuit output is constructed and returned as the result of evaluating procedure **generate-equation**.

The Tester Module

The tester module (Figure 5.7) decomposes the list that was output from the equation generation module to form lists which represent the initial characteristic equation, the circuit inputs, the output variable, and the checkpoint variables. These lists are used to generate tests and deduce new information about the circuit.

```

(((Z--) AX1 D01) ((Z--) A-- (AX0) D01) ((B--) (BX1) (Z--) AX1 (D00))
((B--) (BX1) (Z--) A-- (AX0) (D00)) (CX1 (Z--) D11) (C-- (CX0) (Z--) D11)
((C--) (CX1) Z-- (A--) (AX1)) ((CX1) (C--) BX1 (D01) Z--)
(BX1 (D11) Z-- (A--) (AX1)) (BX1 (D11) (D01) Z--) (CX0 CX1)
((CX1) (C--) B-- (BX0) (D01) Z--) (B-- (BX0) (D11) Z-- (A--) (AX1))
(B-- (BX0) (D11) (D01) Z--) ((B--) (BX1) CX1 (Z--) (D10))
((B--) (BX1) C-- (CX0) (Z--) (D10)) (D10 D11) (D10 Z-- (A--) (AX1))
(CX0 Z-- (A--) (AX1)) (CX0 BX1 (D01) Z--) (CX0 B-- (BX0) (D01) Z--)
(BX0 (Z--) AX1 (D00)) (BX0 (Z--) A-- (AX0) (D00)) (BX0 CX1 (Z--) (D10))
(BX0 BX1) (BX0 C-- (CX0) (Z--) (D10)) (D00 D01) (D00 Z-- CX0)
((C--) (CX1) Z-- D00) (D00 Z-- D10) (AX0 AX1) (AX0 Z-- CX0)
((C--) (CX1) Z-- AX0) (AX0 Z-- D10) (BX1 (D11) Z-- AX0)
(B-- (BX0) (D11) Z-- AX0))

```

Figure 6.26. Characteristic Equation

Test Generation. In the first test of the circuit of Figure 5.2, the input function, $i_1(\underline{x})$, was identically equal to zero. This condition is shown in Figure 6.6 by the statement

The Input Equation is: $0 = 0$.

Hence any test vector was effective. The test vector $(1,1,1)$ was chosen arbitrarily. Using the result of this test, the steps outlined in Chapter 4 were followed to form an updated characteristic equation. The updated characteristic equation is given in Figure 6.27.

```

((B-- (BX0) (D11) (AX1) (A--) Z--) ((BX0) (D11) D00) (AX0 D10)
((BX0) (D11) AX0) ((BX0) (D11) (D01)) ((BX0) (D01) CX0)
((BX0) (D01) D10) (CX0 AX0) (A-- B-- C-- (Z--)) (CX0 D00)
(B-- (BX0) (D01) Z-- (CX1) (C--)) ((BX1) (B--) CX1 (D10) (Z--))
(A-- (AX0) (D00) (Z--) BX0) (A-- (AX0) D01 (Z--)) (D00 D10)
((D10) C-- (CX0) (Z--) BX0) (D11 D10) (D11 (CX0) C-- (Z--))
(D01 D00) (D11 CX1 (Z--)) (CX0 CX1) ((D10) CX1 (Z--) BX0)
((CX1) Z-- (C--) AX0) (Z-- (CX1) (C--) D00) (AX1 AX0)
(AX1 D01 (Z--)) (AX1 (D00) (Z--) BX0) ((AX1) (A--) Z-- D10)
((AX1) (A--) Z-- CX0) ((AX1) (A--) (CX1) Z-- (C--)) (BX1 BX0)
(BX1 (D01) Z-- (CX1) (C--)) (BX1 (D01) D10) (BX1 (D01) CX0)
(BX1 (D11) (D01)) (BX1 (D11) AX0) (BX1 (D11) D00)
(BX1 (D11) (AX1) (A--) Z--) ((BX1) AX1 (D00) A-- C-- (Z--))
((BX1) CX1 (D10) A-- C-- (Z--)) ((BX1) (CX0) (D10) A-- C-- (Z--))
((BX1) (AX0) (D00) A-- C-- (Z--)) ((BX1) (B--) A-- (AX0) (D00) (Z--))
((BX1) (B--) C-- (CX0) (D10) (Z--)) ((BX1) (B--) AX1 (D00) (Z--)))

```

Figure 6.27. An Updated Characteristic Equation

The updated characteristic equation is used to produce the second input equation, $i_2(\underline{x}) = 0$. In this case, the input equation that is derived is given by $((A--\ C--))$ where $(A--\ C--)$ is a single term which is equal to zero. This is illustrated in Figure 6.6 by the statement

The Input Equation is: $A\ C = 0$.

Solutions to the input equation are effective test vectors. As discussed in Chapter 4, one method to solve $i_2(\underline{x}) = 0$ is to form the equation $i'_2(\underline{x}) = 1$; minterms of $i'_2(\underline{x})$ are solutions to the input equation. This is shown as follows:

$$\begin{aligned} AC = 0 &\Leftrightarrow (AC)' = 1 \\ &\Leftrightarrow A' + C' = 1. \end{aligned} \tag{6.1}$$

The minterms of $A' + C'$ are the effective test vectors.

Performing the operations given by (6.1), the diagnostic system transforms the list $((A--\ C--))$ to the list $((((A--))\ ((C--)))$. The list $((((A--))\ ((C--)))$ is a two term list which is equal to one. $((A--))$ and $((C--))$ are distinct terms of the sum-of-products formula. In each of these terms, there is a single complemented literal.⁵ To generate a test vector, the diagnostic system uses the term represented by the first element of the list $((((A--))\ ((C--)))$. This happened to be the term $((A--))$ in this case. Since $A' = 1$, the circuit input A must be equal to 0 to form an effective test vector. Other inputs can be chosen arbitrarily. In the implementation of the diagnostic system, the rule which is followed is to always make an unspecified input a value of 1. Hence, as shown in Figure 6.6, the test vector that was output to the user was specified by

$$\begin{aligned} A &= 0 \\ B &= 1 \\ C &= 1 \end{aligned}$$

⁵As noted previously, a literal in a term is complemented if it is enclosed in parentheses.

Recall that in the first test of the circuit, all inputs were chosen arbitrarily. Following the guideline that unspecified inputs are set to the value of 1, the first test vector that was generated was to set all inputs to 1. The discovery of a heuristic which may make the selection process more intelligent is a subject for further study.

When the list $(())$ is returned as the result of producing $i(\underline{x})$, then the input function is equal to 1.⁶ This is the condition discussed in Chapter 4 which demonstrates that all information deducible from testing has been obtained.

Formation of a New Constraint Using the test vector and the circuit response to the test a new constraint is derived which is used to update the characteristic equation. From Figure 6.6, when the test $A = 0, B = 1, C = 1$ was applied to the circuit, the response of the circuit was 1. From equation (4.48), when the response of a circuit to a test is 1, the new constraint is formed by the equation

$$m_j(\underline{x}) \cdot z' = 0, \quad (6.2)$$

where $m_j(\underline{x})$ is the input test vector, and z is the circuit output variable. Hence, for the given test vector and circuit response, the new constraint is given by:

$$A'BCZ' = 0. \quad (6.3)$$

The structure which represents this new constraint is:

$$((A--) \ B-- \ C-- \ (Z--)).$$

⁶In the sum-of-products list notation, $(())$ is equal to 1.

A new constraint forms new information which is incorporated into the characteristic equation to form an updated characteristic equation. Testing continues until new information cannot be obtained from testing. This condition is noted in Figure 6.7 by the statement

New information cannot be obtained.

At this point, the final characteristic equation is used to determine the function that the circuit is performing as well as the possible fault conditions in the circuit.

The Interpretation Module

The interpretation module (Figure 5.9) uses the final characteristic equation, the list of the circuit checkpoints, and the circuit output variable to determine the actual circuit function and the fault conditions of the circuit under test.

Determination of the Circuit Function. The procedure which determined the actual function of the circuit was easily implemented. Using the procedure outlined in Chapter 4, the final characteristic equation (Figure 6.28), the checkpoints and the circuit output variable are used to determine the actual circuit function. First, all terms which involve checkpoint variables are removed. Since the characteristic equation is in Blake canonical form, elimination of the checkpoint variables is performed simply by removing the terms in which a literal is a checkpoint variable. Performing these operations leaves the structure given in Figure 6.29; this is the exclusive-OR of the circuit inputs with the circuit output as given by equation (4.80). The structure in Figure 6.29 represents the equation

$$az' + cz' + a'c'z = 0. \quad (6.4)$$

This equation may be rewritten as

$$(a + c)z' + a'c'z = 0 \quad (6.5)$$

where $(a + c)$ represents $a(\underline{z})$ as depicted by equation (4.82). Generation of $a(\underline{z})$ is performed by removing the term of the structure of Figure 6.29 which includes the uncomplemented form of \underline{z} , and then removing the (\underline{z}) literal from the remaining terms. The diagnostic system then removes the suffix from the literals and prints out the result as depicted in Figure 6.7.

```
((((BX1 (D11)) ((BX0 (D11)) ((BX0 (D01)) (A-- (Z--))
(CX1) (BX0 BX1) (BX1 (D01)) (AX0) (CX0) (D10) (D00)
(AX1) (C-- (Z--)) ((A--) (C--) Z--)))
```

Figure 6.28. The Final Characteristic Equation

```
((A-- (Z--)) (C-- (Z--)) ((A--) (C--) Z--))
```

Figure 6.29. The Exclusive-OR of the Circuit Inputs and Output

Determination of the Designed Circuit Function. To determine whether the actual circuit function is the same as the designed circuit function, a Boolean equation representing the designed circuit function had to be derived. By (4.82), the actual circuit function which is derived expresses the circuit output as a function of the circuit inputs. However, the original expression describing the circuit was not in this form; the circuit was described as a system of equations representing gates. Therefore, the original expression had to be manipulated to derive an expression which depicts the circuit output as a function of the circuit inputs.

The intermediate format was converted directly to sum-of-products form to attain a single equation which represents the circuit. The structure representing this equation is shown in

Figure 6.30. This equation differs from the single equation derived earlier, because in this case checkpoints are not considered. By (B.62), elimination can be used to eliminate internal nodes from the equation. The resultant of elimination is an equation which is similar in form to (4.80); the structure representing this equation is given in Figure 6.31. This equation is used to form an equation which expresses the output of the circuit as a function of the circuit inputs. After removing the suffixes from the literals, the diagnostic system prints out the result as shown in Figure 6.7.

```
((D-- B--) ((D--) (B--)) (E-- A-- D--) ((E--) (A--))
(F-- C-- D--) ((F--) (C--)) ((F--) (D--)) (Z-- E-- F--)
((E--) (D--)) ((Z--) (E--)) ((Z--) (F--)))
```

Figure 6.30. Sum-of-Products Form of the Circuit Under Diagnosis

```
(((B--) C-- (Z--)) ((C--) (A--) Z--) ((B--) A-- (Z--)) (B-- Z--)))
```

Figure 6.31. Sum-of-Products Form After Elimination of Internal Nodes

Comparison of the Actual to the Designed Function. The actual and designed circuit functions are compared using the exclusive-OR tests which is outlined in Appendix B. Since the equations represented in Figures 6.29 and 6.31 are in $f = 0$ form, the exclusive-OR test may be used. If the result of the XOR is a 0, then the circuits are equivalent; otherwise, the circuits are not equivalent. The result of this test is given in Figure 6.7.

Deduction of Fault Conditions. As is the case with determination of the actual circuit function, the state of the checkpoints is generated using the final characteristic equation. One method for deducing the checkpoint state equation, $G(y) = 0$, as defined by (4.88) is to eliminate

the output variables from the final characteristic equation. The resultant of elimination is an equation which represents the state of the checkpoint variables.

Since the final characteristic equation given in Figure 6.28 is in Blake canonical form, elimination of the output variables is performed by deleting terms which include literals of the output variable. The resulting equation, $G(\underline{y}) = 0$, is represented by Figure 6.32. Solutions of $G(\underline{y}) = 0$ yield the possible faults in the circuit. Using the method outlined in Appendix B, the checkpoint state equation is placed in the form $G'(\underline{y}) = 1$. The minterms of $G'(\underline{y})$ are solutions of the checkpoint state equation. $G'(\underline{y})$ is given in Figure 6.33.

```
((BX1 (D11)) ((BX0) (D11)) ((BX0) (D01)) (AX1)
(CX1) (BX0 BX1) (BX1 (D01)) (AX0) (CX0) (D10) (D00))
```

Figure 6.32. The Checkpoint State Equation

```
((((AX0) (AX1) (BX0) (CX0) (CX1) (D00) D01 (D10) D11)
((AX0) (AX1) BX0 (BX1) (CX0) (CX1) (D00) (D10))
((AX0) (AX1) (BX1) (CX0) (CX1) (D00) D01 (D10) D11))
```

Figure 6.33. The Complemented Form of the Checkpoint State Equation

In this example, $G'(\underline{y})$ consists of three terms representing three possible fault conditions of the circuit. However, although there are three fault possibilities, the condition of certain nodes can be determined with certainty. This can be done by examining the literals of each term of $G'(\underline{y})$. If a literal exists in uncomplemented form in each term, then the fault condition is asserted; likewise, if a literal exists in complemented form, then the associated fault condition is denied. For example, the literal **AX0** exists in complemented form in each of the terms depicted in Figure 6.33; hence by Table 4.1, node **A** is not stuck-at-0. Since **AX1** also appears in uncomplemented form in each term, we can conclude the node **A** is not stuck-at-1. Since node **A** is both not stuck-at-0 and not stuck-at-1, by Table 4.1 we conclude that the state of node **A** is normal. Using the same deduction

process, we can also conclude that node C is normal, and that the two checkpoints associated with node D are not stuck-at-0.

The condition of all nodes which was determined with certainty was output to the user as shown in Figures 6.7 and 6.8. One function that the interpretation module must perform is to associate the checkpoint variables as shown in Figure 6.33 with a particular node in the circuit. The state of a given node is the information that is useful to the user, not the state of the checkpoint variables. The suffixes of the checkpoint variables are used to perform this association. If a checkpoint variable is of the form **NodeX0**, the variable is associated with an input node which does not fan out. Hence, this association is trivial. However, association of a checkpoint variable with a fanout node is more difficult. In a checkpoint variable associated with a fanout node, a number from 0 to 9 was used in place of an X. The fanout branches are labeled from 0 to 9 depending on their occurrence in the intermediate format; the fanout branch node which occurs first is labeled 0, the second is labeled 1, and so on. This is demonstrated in Figure 6.19. To simplify the output of information to the user, the fanout branch is referred to with respect to the gate for which it is an input. The intermediate format of Figure 6.19 is used to make the translation from the number of the fanout branch to the associated gate. The final result of this translation is shown in Figure 6.8. When information is output to the user, suffixes are removed from the nodes; checkpoint nodes associated with fanout branches are associated with a gate.

After information that was determined with certainty was output, the remaining possible fault cases must be output to the user. The way this is performed is to eliminate all common literals from the terms of the equation of Figure 6.33. The resulting equation is given in Figure 6.34. After common literals are removed, each term is interpreted separately to deduce the possible states of faults in the circuit. Table 4.1 is used to deduce the state of the nodes. The output generated by interpreting these three fault cases is shown in Figure 6.8. As noted previously, the circuit response that was fed back to the diagnostic system was predicated on the assumption that node b

of the circuit depicted in Figure 5.2 was stuck-at-0. This was one of the fault conditions projected by the diagnostic system. Examination of the circuit clearly shows that the two other possible fault conditions can cause the circuit to act in the same manner as the fault condition of node b stuck-at-0. Hence, the possible fault conditions are all members of the same equivalence class.

```
((BX0) D01 D11)
( BX0 (BX1))
((BX1) D01 D11))
```

Figure 6.34. The Complemented Checkpoint State Equation with Common Literals Removed

The commented source code in Appendix D gives further explanations which detail how procedures in the diagnostic system work. Study of these procedures as well as a knowledge of deviations from the Revised³ Report on the Algorithmic Language Scheme are essential to implementing modifications to the system as well as porting it to different implementation of the Scheme language.

Summary of Portability Issues

The Scheme language is defined by the Revised³ Report on the Algorithmic Language Scheme. The report defines the primitive procedures which must be included in an implementation of the Scheme language; these procedures are called *essential* procedures [Rees 86:39]. Other procedures are defined with the caveat that they may be included in an implementation but are not mandatory. However, most implementations of the language include a great number of the optional procedures. Additionally, extensions may be included in implementations "provided the extensions are not in conflict with the language reported" in the report [Rees 86:39]. For a complete listing of procedures defined in the Scheme language see the Revised³ Report on the Algorithmic Language Scheme [Rees 86] or *The SCHEME Programming Language* [Dybv 87].

PC Scheme conforms to the Revised³ Report with minor extensions. These extensions include procedures which conform to Common Lisp expressions, operations to make input-output easier than procedures defined in the Revised³ Report, and routines for interfacing with the MS DOS operating system. PC Scheme implements most of the optional procedures defined in the Revised³ Report. In this section all procedures used in the implementation of the diagnostic system which are considered optional or are extensions to the Scheme language are outlined. This information is necessary if the system developed in this project is to be ported to a computer which uses a different version of Scheme.

Optional Procedures. In this section all optional Scheme procedures implemented by PC Scheme and used in the implementation of the diagnostic system are outlined. If the diagnostic system is ported to a computer which uses an implementation of Scheme which does not include the optional procedures described here, then the user must write procedures to replace the optional procedures using the primitive procedures which are available. If the optional procedures used cannot be replicated in function, then procedures in the diagnostic system will have to be rewritten to accommodate this fact. For each construct, the name of construct is given with the function that it performs. These descriptions were taken from the TI Scheme Language Reference Manual [Texas 87b].

File Manipulation Procedures.

- (**open-input-file** *filename*): Opens the file specified by *filename* for input, and returns the port for the file.
- (**open-output-file** *filename*): Opens the file specified by *filename* for output, and returns the port for the file. If the file by the same name exists, it is overwritten.
- (**close-input-port** *port*): Closes an input port; if the port is associated with a file, then the file is closed.
- (**close-output-port** *port*): Closes an output port; if the port is associated with a file, then the file is closed.

String and Character Manipulation Procedures.

- (**char-upcase** *character*): Returns the uppercase form of a lowercase alphabetic character; otherwise returns the character.
- (**make-string** *size character*): Builds a string of characters of length *size* consisting of the specified *character*.

Logical Operators.

- (**and** *exp* ...): Connects expressions (*exp*) to form a compound predicate.
- (**or** *exp* ...): Connects expressions (*exp*) to form a compound predicate.

Numeric operators.

- (**expt** *number exponent*): Returns the value of *number*^{*exponent*}.

Special Forms.

- (**do** ((*var value update*)...) (*test exp*) *stmts*): Provides the capability to perform iteration. First the set of variables *var* are assigned initial values specified by *value*. After each iteration, the value of *var* is updated using the expression denoted by *update*. After *var* is updated, it is checked by *test*. If the test results in a true condition, then the value returned by the **do** statement is given by *exp*. Otherwise, the set of expressions given by *stmts* is evaluated sequentially, and the cycle iterates.
- (**let*** ((*var form*)...) *exp1 exp2* ...): Evaluates expressions in an extended environment. Variables designated by *var* are local variables created for use within the **let*** environment. The values assigned to *var* are given by the expressions denoted by *form*. The expression *form* may use the value of preceding local variables during evaluation. After all variables are assigned values, then the expressions *exp1 exp2* ... may use *var* when executing. The value returned by **let*** is the value of the last expression in the sequence of expressions *exp1 exp2* ...

MS DOS Interfacing Procedures. In this section, procedures used to interface with the MS DOS operating system are described. If the diagnostic system is used on a computer which uses a different operating system, then the pertinent operating system interface procedures must be substituted for those described here.

- (**dos-delete** *filename*): Deletes the file specified by *filename*; returns an error if the file does not exist.
- (**dos-rename** *old-filename new-filename*): Changes the name of the file specified by *old-filename* to the name given by *new-filename*; returns an error if the file does not exist.
- (**file-exists?** *filename*): Returns true if the file given by *filename* exists; otherwise, returns false.

Scheme Extensions. PC Scheme provides a set of extensions to the Scheme language that are not listed in the Revised³ Report. As with optional procedures, if the diagnostic system is ported to a computer which uses an implementation of Scheme which does not include the

extended procedures, then the user must write procedures to replace the extended procedures using the primitive procedures which are available. Throughout the implementation effort, extended procedures were avoided unless it was obvious that similar procedures could be created using a combination of essential primitive procedures.

Object Operators.

- (*atom? object*): Returns true if *object* is not a pair; otherwise, returns false.

Numeric Operators.

- (*1+ number*): Returns the result of adding 1 to *number*.
- (*-1+ number*): Returns the result of subtracting 1 from *number*.

Input-Output Procedures.

- (*read-line {port}*): Reads a sequence of characters from *port* from the current position up to the next end-of-line or end-of-file and returns it as a string. *port* may specify a file or the console (keyboard). The alternative to using this procedure was to process a character at a time.
- (*writeln obj1 ...*): Evaluates objects specified by *obj1* ... and prints the results of the evaluation to the current output port. The current output port may be either a file or the console (screen). Then, a newline is output.
- (*princ exp {port}*): Evaluates the expression *exp* and prints its value to *port*.

Summary

In this chapter, the implementation of the diagnostic system was discussed. An introduction to the Scheme programming language and the PC Scheme environment was presented. After introducing an example of a diagnostic session, the transformation of data as it flows through the diagnostic system was illustrated. Design decisions made throughout the implementation process were highlighted. Finally, procedures in PC Scheme used in the implementation of the diagnostic system which are either optional or extensions of the Scheme programming language were outlined.

The Scheme programming language was a useful medium for implementing the diagnostic system. Because equations were manipulated symbolically, experimentation during system im-

plementation that would have been prohibitive with a conventional programming language was feasible. For example, the use of the $f = 0$ form of an equation rather than the $p = 1$ form was found to be advantageous due to experimentation during the implementation process. Hence, the use of the Scheme language for system implementation facilitated better design decisions.

VII. Results

This chapter discusses the results of diagnosing circuits with the diagnostic system. The complexity of the algorithm developed in this project is examined. In light of the complexity, limitations of the utility of a PC-based diagnostic system are discussed. Different circuits were used to validate the operation of the system.

Algorithm Complexity

The complexity of the algorithm developed in this project is related to the growth in the number of terms which represent a function as the number of variables of the function increases. The primary operation which produced these terms was the generation of the Blake canonical form of a function—the disjunction of all of the prime implicants of the function. The processing time required to produce the Blake canonical form dominated all other operations which were performed. Considering the total processing time of all steps of the diagnostic algorithm, approximately ninety percent of all processing was devoted to the production of the Blake canonical form.

For n variables, the possible combinations of variables which can form terms is $3^n - 1$. Each variable occurs in a term in one of three forms: uncomplemented, complemented, or not at all. Considering all combinations of variables in these three forms, 3^n terms are formed. However, the “term” formed by the case of all variables not appearing is subtracted to yield the figure of $3^n - 1$. Thus, an upper bound on the number of terms in a sum-of-products formula which may represent an n -variable Boolean function is $3^n - 1$. In the circuit of Figure 5.2, the initial characteristic function is a function of 12 variables. The circuit has three inputs, one output, and four checkpoints; each checkpoint introduces two variables to the characteristic function. Thus, the number of possible terms in a sum-of-products formula representing the initial characteristic function is:

$$3^{12} - 1 = 531440. \quad (7.1)$$

Typically, however, the total number of terms in a formula which can represent a given function is much less than $3^n - 1$. The set of all terms in such a formula can be reduced to a smaller set of terms whose disjunction represents the function. One example of such a set is the Blake canonical form of a function. The algorithm developed in this project is based on the production of the Blake canonical form. Hence, the complexity of the diagnostic algorithm is of the same order as the complexity of generating the prime implicants of a function. Studies have been conducted which examined the bounds of the complexity of prime implicant generation. One of the earliest studies was done by Dunham and Fridshal; they developed a function which gives a lower bound of the maximum number of prime implicants with respect to the number of variables n of the function [Dunha 59]. Igarashi developed a revised lower bound for the maximum number of prime implicants of a function. He determined that the lower bound developed by Dunham and Fridshal underestimated the lower bound of the maximum number of prime implicants. In lieu of their lower bound, he developed a recursive procedure for determining the lower bound of the maximum number of prime implicants with respect to the number of variables n of the function [Igara 79].

More important to the diagnostic algorithm developed in this project is an upper bound on the maximum number of prime implicants of a Boolean function. We would like to know the greatest number of prime implicants of a function of n variables—an upper bound on the size of the Blake canonical form of a function. Defining a function $g(n)$ which is the maximum number of prime implicants of a Boolean function of n variables, Chandra and Markowsky developed an upper bound for $g(n)$ [Chand 78:9]. They state that "g is a measure of the complexity of boolean functions on n variables" expressed in sum-of-products form [Chand 78:10]. The bound which they developed is

$$\binom{n}{\lfloor (2n+1)/3 \rfloor} 2^{\lfloor (2n+1)/3 \rfloor}, \quad (7.2)$$

where

$$g(n) \leq \binom{n}{\lfloor (2n+1)/3 \rfloor} 2^{\lfloor (2n+1)/3 \rfloor}. \quad (7.3)$$

[Chand 78:10]

This bound is $O(3^n/\sqrt{n})$ [Chand 78:10]. The complexity of the diagnostic algorithm developed in this project shares the same upper bound.

One of the primary operations used to form the Blake canonical form is consensus. Consensus in the Boolean domain is the analog of the resolution of propositions in mechanical theorem proving—the complexity of each is the same*. Galil derived an expression of the lower bound of the complexity of resolution of propositions. He showed that

$$\text{Comp}_{Reg}(n) \geq 2^{n/72(\log n)^2} \quad (7.4)$$

where n is the number of variables, and $\text{Comp}_{Reg}(n)$ is the complexity of regular resolution [Galil 75:10]. This expression forms a lower bound for the complexity of performing consensus when generating the Blake canonical form.

The diagnostic algorithm based on the Blake canonical form of a function is clearly of exponential complexity. The bounds on the number of prime implicants of a function indicate the space complexity of the algorithm; the lower bound of the consensus operation yields a measure of the time complexity. This result is not unexpected given that the complexity of generating a test to detect a single stuck-at fault is an NP-complete problem [Abadi 85:9]. However, the bounds of the maximum number of prime implicants of an n -variable Boolean function represents a theoretical worst-case function. In practice, the number of prime implicants of an n -variable function which represents a combinational circuit is far less than the maximal number of prime implicants which comprise the theoretical worst-case n -variable function.

The initial characteristic function which represents a circuit under diagnosis is in Blake canonical form—the disjunction of all of the prime implicants of the function. For all examples used in this project, the number of terms of the initial characteristic function is small relative to the upper bound of the theoretical maximum number of prime implicants for a Boolean function of the same number of variables. Table 7.1 depicts results for several circuits diagnosed in this project. Listed are the number of variables which form the initial characteristic function, the number of terms (prime implicants) which make up the function, and the theoretical upper bound of the maximum number of prime implicants of a Boolean function of the same number of variables.

Circuit	No of Vars	No of Terms	Upper Bound of Max No Terms
Figure 7.4	7	15	672
Figure 7.1	8	45	1792
Figure 3.1	10	17	15,360
Figure 5.2	12	36	126,720

Table 7.1. Comparison of Actual Number of Prime Implicants to the Theoretical Case

While the data derived from examples do not prove anything about the complexity of the approach—the theoretical analysis yields this information—the results gathered in this project indicate that the algorithm developed in this project is not altogether unfeasible. It is possible that the manner in which a Boolean function representing a circuit is formed yields a bound on the number of prime implicants that is problem-specific. For example, in Table 7.1 the circuit of Figure 7.1 yielded a characteristic function which had more prime implicants than a characteristic function of a greater number of variables. Examination of the circuits referenced by Table 7.1 bears out the fact that this circuit included an XOR gate, whereas the other circuits did not. Hence, the number of prime implicants which form the characteristic function appears to be dependent on the composition of the gates of a circuit as well as circuit topology. Further research must be performed to develop the bounds of the number of prime implicants of the characteristic function as a function of circuit composition and topology, as well as the number of variables of the function. It is possible that

such a bound, if it exists, is less than the bounds developed for the maximum number of prime implicants of a theoretical n -variable Boolean function.

Circuit Diagnosis

After the diagnostic system was implemented, it was used to diagnose several circuits to validate its operation. A number of circuits were used to insure the system could diagnose faults in circuits of different topology and gate composition. In no case did the diagnostic system provide inaccurate results regarding the possible state of faults in the circuit.

The method used in the validation process was as follows:

1. Generate a set of equations or a VHDL representation to describe a given circuit.
2. Assume that a particular fault condition has occurred in the circuit.
3. Use the circuit description as an input to the diagnostic system.
4. For each test vector generated by the diagnostic system, determine the response of the circuit under the assumed fault condition. Input to the diagnostic system the faulty circuit's response.
5. Compare the output of the diagnostic system to the assumed fault condition to insure that the system diagnosed the proper fault.

About ten circuits were diagnosed using the diagnostic system. In many cases, the same circuit was used with different assumed fault conditions. Figures 3.1 and 5.2 depict two circuits which were diagnosed with the system. A complete diagnostic session using the circuit of Figure 5.2 is given in Chapter 6; the circuit was assumed to have a stuck-at-0 fault on node **b**.

Figure 7.1 illustrates another circuit diagnosed by the system. In this circuit, it was assumed that a s-a-1 fault occurred on the branch of node **b** which is an input to the XOR gate and a s-a-0 fault occurred on the branch of node **b** which is an input to the OR gate. The results of diagnosis of this circuit are given in Figures 7.2 and 7.3.

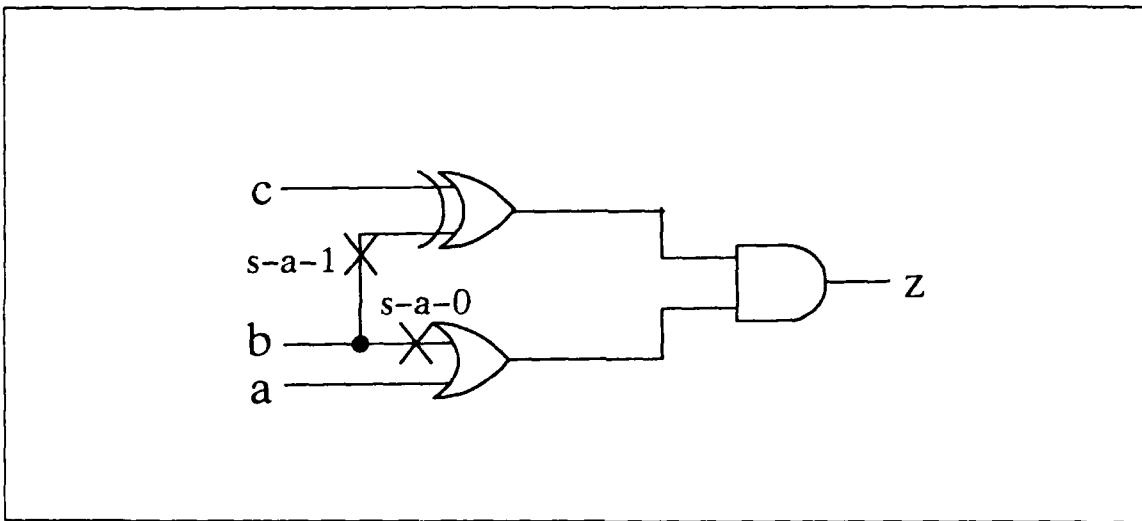


Figure 7.1. Diagnosis of Circuit with XOR Gate and Two Faults

Detection of Redundancy

One application of the diagnostic system mentioned in Chapter 4 was for redundancy detection within circuits. In this section, an example is given to demonstrate this capability.

In this application, the correct output is fed back to the diagnostic system. However, the diagnostic algorithm is guaranteed to produce a list of the faults that may exist in the circuit. Hence, the procedure produces a list of undetectable faults if the correct outputs are fed back. The circuit of Figure 7.4 is a redundant circuit that was used as an input to the diagnostic system. For all test vectors generated by the diagnostic system, the correct output was fed back as the result of the test. The results of the diagnosis are given in Figures 7.5 and 7.6.

In spite of the fact that the correct outputs were fed back to the diagnostic system, the diagnostic system produced four sets of possible faults for the circuit. Clearly, since there are undetectable faults, there is redundancy in the circuit. Hence, the diagnostic system detected redundancy in the circuit. Additionally, examination of the possible fault locations gives an indication of the location of the redundant nodes in the circuit.

***** Results *****

The function that the circuit was designed to perform is:

$$Z = B C' + A B'C$$

The function that the circuit is performing is:

$$Z = A C'$$

The actual circuit IS NOT equivalent to the designed circuit.

**** The following information is certain about the circuit ****

Input nodes (which do not fanout) that are normal:

A
C

Input nodes (which do not fanout) that are stuck-at-0:

--none--

Input nodes (which do not fanout) that are stuck-at-1:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-0:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-1:

--none--

Fanout nodes that are normal:

--none--

Fanout nodes that are stuck-at-0:

Node B of gate: $F = A + B$

Fanout nodes that are stuck-at-1:

Node B of gate: $E = B ! C$

Figure 7.2. Diagnosis of Circuit in Figure 7.1

```
Fanout nodes that are NOT stuck-at-0:  
--none--  
Fanout nodes that are NOT stuck-at-1:  
--none--  
System Performance Metrics:  
The number of tests run was: 5  
The number of possible tests was: 8  
The performance ratio is: 0.625
```

Figure 7.3. Diagnosis of Circuit in Figure 7.1 (cont.)

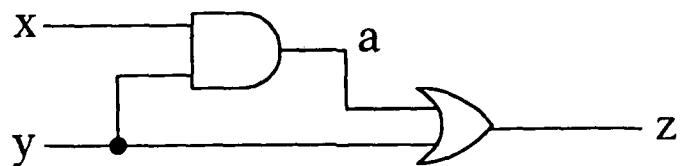


Figure 7.4. A Redundant Circuit

***** Results *****

The function that the circuit was designed to perform is:

Z = Y

The function that the circuit is performing is:

Z = Y

The actual circuit IS equivalent to the designed circuit.

**** The following information is certain about the circuit ****

Input nodes (which do not fanout) that are normal:

--none--

Input nodes (which do not fanout) that are stuck-at-0:

--none--

Input nodes (which do not fanout) that are stuck-at-1:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-0:

--none--

Input nodes (which do not fanout) that are NOT stuck-at-1:

--none--

Fanout nodes that are normal:

--none--

Fanout nodes that are stuck-at-0:

--none--

Fanout nodes that are stuck-at-1:

--none--

Figure 7.5. Diagnosis of a Redundant Circuit

```

Fanout nodes that are NOT stuck-at-0:
--none--

Fanout nodes that are NOT stuck-at-1:
Node Y of gate: Z = A + Y
**** One of the following cases holds for the circuit ****
**** Case #1 ****
Input node X is not stuck-at-1.
Node Y of gate: Z = A + Y is not stuck-at-0.
Node Y of gate: A = X * Y is not stuck-at-1.

**** Case #2 ****
Input node X stuck-at-0.
Node Y of gate: Z = A + Y is not stuck-at-0.
Node Y of gate: A = X * Y is not stuck-at-0.

**** Case #3 ****
Input node X is not stuck-at-0.
Node Y of gate: Z = A + Y is not stuck-at-0.
Node Y of gate: A = X * Y is not stuck-at-1.

**** Case #4 ****
Input node X is stuck-at-1.
Node Y of gate: A = X * Y is not stuck-at-0.
Node Y of gate: A = X * Y is not stuck-at-1.

System Performance Metrics:
The number of tests run was: 3
The number of possible tests was: 4
The performance ratio is: 0.75

```

Figure 7.6. Diagnosis of a Redundant Circuit (cont.)

System Limitations

Despite the success of the diagnostic system, the system was inefficient for diagnosis of all but the smallest circuits. For example, the diagnostic system exhausted available memory when diagnosing the circuit illustrated in Figure 2.7. Hence, diagnosis could not be completed for that circuit.

Other circuits with approximately seven checkpoints or greater could not be diagnosed with the diagnostic system. The memory available in a PC-based system severely restricts the size of a circuit that may be diagnosed; a PC-based system is insufficient given the complexity of the problem. A subject for further work is to port the diagnostic system to a virtual-memory environment. Once the system is operational in a virtual-memory environment, a variety of circuits should be diagnosed to make a judgement regarding the maximum practical size of circuit that can be diagnosed with this system and to better compare this system to other methods.

Although available memory may be the limiting factor with respect to the size of circuit that may be diagnosed, implementation details may be a primary cause for the inefficiency of the system. The Scheme language is properly tail recursive; however, procedures were not always written to take advantage of this. In addition, no studies were conducted regarding the data structure used to represent a Boolean formula in this implementation. The data structure that was used may not have been the most efficient. A prime example of an implementation issue which had a great impact on the efficiency of the system was the use of the $f = 0$ versus the $p = 1$ form when generating the characteristic equation; other issues which were not studied may be just as critical.

VIII. Conclusions and Recommendations

Summary

Fault diagnosis is an important problem in digital design and testing. It is vital to insure that circuits in automated systems perform correctly, and if they do not, to be able to isolate and correct faults that have occurred. However, there exist deficiencies in currently-available diagnostic systems. Such systems typically are based on restrictions or assumptions which limit their capability. In most current systems it is assumed that only a single fault may exist in a circuit, or fault detection is performed but not fault location. The goal of this thesis was to develop a diagnostic system which overcomes many of the limitations which restrict current systems. Using a circuit model based on Boolean equations and a reasoning method called Boolean reasoning, a diagnostic algorithm was developed which adaptively locates multiple stuck-at faults in combinational circuits.

A fault model was developed to mathematically model the state of faults in the circuit. The circuit description and the checkpoint fault model are processed to derive a single Boolean characteristic equation. The characteristic equation is used to generate effective test vectors which are used as inputs to the actual circuit under test. After a test vector has been input to the circuit, the output is observed. The state of the circuit output is input to the diagnostic system which uses it to derive new information about the circuit under test and to update the characteristic equation. The new information which is developed places constraints on the process of test generation; hence, a set of tests produced by the system is near-minimal. Tests are conducted repetitively until it is determined that further information cannot be derived from testing, at which point the diagnostic system determines the function that the actual circuit is performing in addition to the nature and location of faults in the actual circuit.

After development of the diagnostic algorithm, a software architecture was developed for the system. Defined in this architecture were the functions of modules in the system, user interfaces, and data to be passed between the modules. A circuit description which can be input to the diagnostic

system can take one of two forms: Boolean equations or statements in the VHSIC Hardware Description Language (VHDL). The output of the system includes equations which describe the actual and designed function of the circuit under test, a listing of all node conditions which can be determined with certainty, and a listing of all possible fault conditions. The locations of faults are determined to within irreducible equivalence classes.

The diagnostic system was implemented with the Scheme programming language. The use of a symbolic language facilitated experimentation throughout the implementation process; such experimentation would not have been feasible using a conventional programming language. A variety of circuits were diagnosed successfully using the diagnostic system; however, the limitations of a PC-based implementation severely restricted the size of circuits that could be diagnosed.

Assessment

The diagnostic system developed in this project overcomes many of the limitations of systems developed previously. The system can adaptively locate multiple stuck-at faults in combinational circuits. Other applications include generation of fault detection test sets, detection of circuit redundancy, and evaluation of fault-detection test sets. The use of constraints to guide the test vector generation process insures that a near-minimal test set is generated. The system does not require processing which is necessary in other techniques. A transformation of the circuit representation is not required. A masking analysis does not have to be performed to insure that all faults are covered by the test set. Additionally, a priori fault enumeration is not required.

The implementation of the system can accept either Boolean equations or VHDL descriptions to represent the circuit under test. The output of the diagnostic system is the function that the circuit is performing and the location of faults to within irreducible equivalence classes.

The circuit and fault models used by the diagnostic algorithm impose several limitations on the system. Diagnosis of the circuit only can be performed at the gate level and diagnosis

is limited to classical faults. Additionally, the algorithm was restricted to handling single-output combinational circuits. Another limitation of the algorithm is that it cannot detect an erroneous model for the circuit under test.

The implementation of the diagnostic system imposes restrictions on the circuit under test. Because of the manner in which fanout branches are handled, a fanout node may have no greater than ten fanout branches. The memory available in a PC-based system restricts the size of a circuit that may be diagnosed. Circuits with greater than seven checkpoints could not be diagnosed because of memory limitations. This limitation is due to the exponential time-space complexity of the diagnostic algorithm. Another limitation of the system is that a file which describes the circuit under test may have no greater than 16383 characters.

Conclusions

This work has successfully employed Boolean reasoning in the task of adaptively locating multiple faults in digital circuits. A diagnostic system was designed and implemented that can locate faults in a single-output combinational circuit. The system accepts either a set of Boolean equations or VHDL statements which describe the circuit; it uses the structural information about the circuit to generate tests which yield further knowledge about the state of the circuit; after the completion of testing, it produces an equation which describes the function of the faulty circuit as well as the locations of faults to within irreducible equivalence classes. All tests generated by the system are effective and knowledge is used to guide the input-output experiment; hence, the set of tests produced by the system is near-minimal.

The symbolic programming language Scheme is extremely useful as an implementation medium for Boolean processes. The ease with which equations can be represented symbolically facilitated experimentation during system implementation that would have been prohibitive with a conventional programming language. Perhaps the combination of Boolean reasoning and the Scheme language can be applied to other problems that have not been solved in the area of digital design and testing.

Despite the success of this project, the model and reasoning method are inefficient for diagnosis of all but the smallest circuits. On a personal computer with 640K of memory, the system exhausted available memory on circuits with greater than six or seven checkpoints. Diagnosis of a VLSI-level circuit with the diagnostic system is unthinkable at this point. A personal computer is insufficient for the implementation of an algorithm with exponential time-space complexity.

The capability of the system may be limited by implementation details. Different ways of implementing algorithms have a great impact on the size of circuit that can be diagnosed. The clearest example of this in this project was the formation of the characteristic equation representing the circuit. The derivation of the $p = 1$ form of the circuit took significantly longer than the $f = 0$ form. Furthermore, the terms of the formula derived in the $p = 1$ form had significantly more literals than terms of the $f = 0$ form. It was important to use the $f = 0$ form in this case because the production of a formula with fewer terms and literals takes less memory space, hence efficiency of the system is improved. Although this particular issue was studied in depth, other implementation details were not studied with respect to efficiency.

Recommendations

This project lays the groundwork for further work in fault diagnosis in general and for improvement of this approach in particular. Topics for study include improvement of the existing

system, extension of the circuit model and algorithm, and exploration of new models and reasoning methods.

Improvement of Existing System. The diagnostic system implemented in this project can be improved in a number of ways. These include integrating features not currently incorporated in the system, porting the system to a larger machine, revising routines to maximize efficiency, devising and implementing heuristics for test generation, and allowing the input of structural VHDL descriptions into the system.

A number of ideas were mentioned which can easily be integrated into the existing diagnostic system. With a minimum of effort, the system can be extended to perform automatic test pattern generation for a given circuit. When the diagnostic system generates a test, the correct circuit output could be fed back to it. The system would continue to generate tests until enough information is available to validate that checkpoints representing detectable faults are normal. The resulting test set would be near-minimal. The correct circuit output guides the system to generate effective tests: when several tests are generated at once, however, there is no guarantee that the test chosen insures that the resulting test set is minimal. The cost of generating the correct circuit output is negligible relative to that of generating the test.

The system should be modified to allow the user to designate the points in the circuit that he believes are faulty. The system would then generate the tests required to determine the states of these designated test points. Thus, tests can be generated to detect specific faults. If the number of these points is small, the system can probably generate tests for moderate-sized circuits. The characteristic equation of the circuit gets larger as a function of the number of primary inputs and test points. However, while an extra input introduces a new variable, a test point causes the addition of two variables to the equation. Thus, the system equation grows multiplicatively with the number of test points. Consequently, if the number of test points is kept low, the system could be used to generate tests for fault detection in larger circuits.

Another feature which can be implemented is the evaluation of a pre-computed test set. A pre-computed test set can be evaluated by the system by adding to the characteristic equation each test with the respective correct circuit response. This is done in the same way that a test and response are added to the characteristic equation in adaptive testing. After all tests and responses have been assimilated into the characteristic equation, the system can determine whether there exist effective test vectors which are not a member of the test set.

Before further work is done on the current diagnostic system, it should be ported to a virtual-memory computing environment to analyze the utility of the approach for medium- to large-sized circuits. Support routines must be written to make existing code compatible with the Scheme environments available on large-scale machines. These modifications are minor in scale. Once the system is operational on a larger machine, a number of circuits should be diagnosed to obtain comparisons of this approach with other methods. Then the system should be analyzed to evaluate whether the current approach is practical.

It is possible that the implementation of the algorithm is causing a large part of the current inefficiency of the system. Implementations details should be studied further and modified if required. Scheme is properly tail-recursive; however, procedures in the system were not always written to take advantage of this efficiency. Additionally, no studies were conducted regarding the data structures used in the implementation—the choices made may not have been optimal.

An idea which would make the system more efficient is the integration of heuristics into the diagnostic process. For example, the system could start with a subset of the circuit checkpoints. The characteristic equation could be generated using this subset. Once a checkpoint is validated to be normal through testing, another checkpoint could then be added to the characteristic equation. Furthermore, when a checkpoint is verified to be normal, all checkpoints between it and a primary output must also be normal by path sensitization. Thus, all of these checkpoints could be eliminated from consideration. This type of iterative approach would limit the combinatoric explosion

engendered by the checkpoints. A heuristic could be developed which shows how to choose the initial subset of the circuit checkpoints. Also, heuristics may exist which can better guide the test generation process.

Another way to improve the existing system is to allow the use of structural VHDL circuit descriptions. The current implementation constrains the user to a restricted form of dataflow description—in essence a “flat” description of the circuit. Structure is then inferred from this description. This is not appropriate for large circuits, because a circuit is typically described hierarchically. Thus, a mechanism should be devised to convert structural view circuit descriptions to the representation required by the diagnostic system—such as the intermediate form developed in this project. A true parser was not used in this project; the integration of structural VHDL would necessitate the development of such a parser. An augmented transition network parser is one possible approach [Tanim 87:349]. Furthermore, a sophisticated file-handling mechanism would have to be developed to allow the use of VHDL descriptions which are stored in different files.

Extension of the Model and Algorithm. The current algorithm was developed to diagnose classical faults in single-output combinational circuits. This limits the utility of the procedure for circuits typically found on integrated circuit chips. Circuits may be combinational or sequential; they normally have multiple rather than single outputs; and they may have nonclassical as well as stuck-at faults. Furthermore, the number of variables introduced into the characteristic equation severely limits the applicability of this system for large circuits. Means of limiting the number of variables should be explored. It may be possible to modify the circuit model and the algorithm used by the system to accommodate multiple-output sequential circuits as well as nonclassical fault diagnosis.

Multiple-output circuits are extensions of single-output circuits. The test generation algorithm as well as the result interpretation procedure should be modified to generalize the system to handle an arbitrary number of outputs. However, the diagnosis of sequential circuits is a far

more complex task. Research should be done to develop algebraic techniques that can be applied to sequential circuit diagnosis. Perhaps the best approach would be to blend other techniques with algebraic methods to locate faults in sequential circuits.

Another restriction of the current system is that diagnosis is limited to classical faults. Other fault models should be integrated into the system. This could be done in a number of ways. One technique is to transform a switch-level CMOS circuit into a logic-gate equivalent network and generate tests for the equivalent circuit. Tests for stuck-at faults in the equivalent circuit have been shown to detect stuck-open and stuck-on faults in the switch-level circuit. Hence, the system should accept a transistor-level circuit description, transform it into an equivalent logic-level circuit representation, and diagnose the equivalent circuit. Faults located in the logic description would map into transistor-level faults. If bridge faults are of concern, then an algebraic model could be developed to facilitate generation of tests to detect and locate these faults.

Since a limitation of the diagnostic system is the growth of the number of terms in the characteristic equation as the number of checkpoint variables gets large, one area of study would be to use a different model which would yield fewer variables. Cha's definition of prime faults is an example of one possible circuit model. Another way to limit the number of variables in the characteristic equation would be to devise a method to partition a circuit. A circuit would be partitioned into testable subcircuits. Each subcircuit would be tested in turn; tests would be generated to determine whether or not a subcircuit is faulty. Test points, located at the inputs and outputs of a subcircuit, would be the only points which would cause the generation of variables in the characteristic equation. This type of hierarchical testing may be the most practical approach to testing complex circuits.

New Models and Reasoning Methods. Given the exponential growth of equations in Boolean-based models as the number of variables increases, a different approach to fault diagnosis may be useful. Another type of model and reasoning method may minimize the number of variables

as well as be better suited for hierarchical diagnosis of circuits. Other methods that have been successfully used for fault location include line-deduction methods and approaches based on artificial intelligence. Of these, artificial intelligence approaches are the most promising.

The model used in this system constrains diagnosis to gate-level components of a circuit. Line-deduction fault location methods have the same limitation. However, artificial intelligence approaches have been shown to be useful for diagnosis at varying levels of abstraction. Logic-based models have been used to describe and diagnose gate-level components as well as functional-level elements of digital systems. Inherent in these approaches is the ability to mix representations at differing levels of abstraction within a single circuit description. This is useful, because portions of a circuit which are not being diagnosed or which are known to be functional can be modeled by a higher-level view while the part of the circuit that is suspected to be faulty can be modeled structurally. In essence, the section of the circuit that is not in the process of being diagnosed is collapsed; not having to represent all gates in the circuit would make a diagnostic system more efficient.

Current diagnostic systems are typically based on generation of tests to detect and locate particular types of faults, e.g., stuck-at faults. Models based on artificial intelligence are not based on assumptions of specific types of faults. These types of models depict circuit components as predicates; a predicate which is shown to be false indicates a faulty component. The type of fault is irrelevant in this approach, only the fact that an element does not work properly. This example is typical of the flexibility intrinsic to AI techniques.

It is widely recognized that currently existing diagnostic systems are insufficient to generate tests to detect and locate faults in VLSI systems. Given this situation, research in new techniques for fault diagnosis will necessarily continue. Artificial intelligence approaches should be explored to develop an adaptive diagnostic system for VLSI-class circuits.

Appendix A. Terminology in Fault Diagnosis

Adaptive experiment: An experiment in which the choice of test vectors is based on the responses of a circuit to previous test vectors [Lala 85:51].

Architecture body: The architecture body describes the function of the VHDL design entity. Hence, the architecture body "specifies the relationships between the inputs and outputs of a design entity" [IEEE 88:1-6].

Behavioral VHDL: The view of VHDL which is used to algorithmically describe the function of a circuit or component. This form would not necessarily embody the structure of a design, rather it describes how it acts.

Bridging fault: A logical fault in which two lines are shorted together.

Characteristic equation: An equation which represents the function and fault conditions of a circuit, denoted by $\Phi(\underline{x}, \underline{y}, z) = 0$. The term "characteristic" was used in a similar manner by Cerny in discussing characteristic functions of a circuit [Cerny 76].

Characteristic function: The function $\Phi(\underline{x}, \underline{y}, z)$ which represents the function and fault conditions of a circuit. Setting this function equal to 0 yields the characteristic equation.

Checkpoints: All primary inputs which do not fanout and all fanout branches in a combinational network. Developed by Bossen and Hong to simplify the process of generating test sets to detect multiple faults.

Checkpoint variables: The variables which represent the state of the checkpoints. Derived from the variables that Poage developed to represent the state of each line in the circuit.

Classical fault: Stuck-at fault.

Combinational circuits: Circuits whose outputs are dependent solely on their current input values, i.e., circuits whose outputs are totally independent of previous input values.

Constraint propagation: A problem-solving approach in which the state of variables is limited by restrictions placed on them. These restrictions are called constraints. After a sufficient number of constraints are gathered, a given variable's value may be derived. Once this occurs, variables dependent on this variable are in turn limited in value. This is called propagation of constraints. [Tanim 87:124].

Constraint suspension: The method used by Davis to deduce which component is acting in a manner inconsistent with its design description. In this method, the intended behavior of the system under test is a collection of constraints, or constraint network, in which each constraint corresponds to the expected behavior of one of the system components. These constraints are derived from the initial component descriptions. During testing, constraints are removed from the collection of constraints until one retraction leaves the constraint network in a consistent state [Davis 85:518].

Dataflow VHDL: The view of VHDL which describes a circuit or component by a collection of VHDL signal assignment statements.

Design entity: "...the primary hardware abstraction in VHDL" [IEEE 88:1-1]. Any part of a design which has a specified function and interface can be represented as a design entity in VHDL. This entity can represent an entire design, a circuit, subcircuit, or a specific component. Each design entity in VHDL is defined by two items, an entity declaration and an architecture body [IEEE 88:1-1].

Detectable faults: Faults whose existence can be determined by a diagnostic experiment. Redundant faults are not detectable.

Effective test: A test in which the outcome is not predictable. Only under this circumstance can new information be derived from the test [Genes 84:423].

Entity declaration: The entity declaration is the portion of a design entity which describes the interface between a VHDL design entity and the environment in which it is instantiated [IEEE 88:1-1].

Equivalence class: A set which includes all equivalent faults. A test which detects one fault in an equivalence class detects all faults in the equivalence class. Also called a fault class.

Equivalent circuits: Two circuits are said to be equivalent if their input-output behavior is indistinguishable.

Equivalent faults: Faults which have the same effect on the input-output behavior of a circuit.

Essential procedures: Primitive procedures which must be included in an implementation of the Scheme language as defined by the Revised³ Report on the Algorithmic Language Scheme. [Rees 86:39]

Experiment: The application of a test set to a circuit and the observation of its reaction. If the observed outputs differ from a pre-computed set of expected outputs, then a fault was detected in the circuit.

Fanout: A point at which several lines diverge from a common originating line. The originating line is called the fanout stem, the others the fanout branches.

Fanout branch: A line diverging from a fanout stem.

Fanout stem: The originating line at a fanout.

Fanout-free circuit: A circuit without fanouts.

Fault: Any physical discrepancy in a circuit [Fujiw 85, Lala 85].

Fault collapsing: A process in which faults are combined based on a study of the circuit to be diagnosed [Scher 72:859]. Thus, tests are developed for classes of faults rather than individual fault cases.

Fault coverage: The percentage of the total number of possible faults in the circuit that a test set may detect [Fujiw 85:17].

Fault detection: The determination that faults exist in a circuit.

Fault diagnosis: The process of determining whether faults exist in a system (fault detection) or locating those that do (fault location) [Abram 80, Lala 85, Solan 86]. Some authors consider fault detection to be distinct from fault diagnosis, equating fault diagnosis and fault location [Fujiw 85, Roy 74].

Fault dictionary: A table in which the specific faults detected by the test vectors in a test set are enumerated. The response of a circuit to the test set is compared to the entries of the dictionary to determine which faults were detected by the test set.

Fault location: The localization of faults in a circuit [Abram 80:452]. Some authors equate fault location with fault diagnosis [Fujiw 85, Roy 74].

Fault masking: A fault, f , masks another fault, f' , for a given test vector, when f' would normally be detected by the test vector if it occurred as a single fault, but would not be detected by the same test vector if both f and f' occurred simultaneously [Fridr 74:855].

Fault model: A representation of the type of faults that may occur in a circuit. The most common is the stuck-at fault model. A fault model is used to generate diagnostic tests for a circuit.

Headlines: A line "that drives a gate that is part of a reconvergent fanout loop" [Kirkl 88:52]. Additionally, a headline must be an output node of a gate in which the predecessor gates are not part of any fanout loop.

Image logic network: The gate-level view of the circuit which is the result of a transformation from a transistor-level view to a gate-level circuit description [Roth 84:59].

Intermittent faults: Temporary faults that appear on some regular basis.

Irredundant circuit: A nonredundant circuit.

Irredundant faults: Faults which are not redundant.

Logical faults: Faults which cause "the logical function of a circuit element (or elements) or an input signal to be changed to some other function" [Breue 76b:15].

Masking analysis: An analysis in which all cases in which one fault masks another for a given test set is determined.

Non-logical faults: All faults that are not logical faults. Also called parametric faults.

Nonredundant circuit: A circuit which is not redundant. Also called an irredundant circuit.

NP-complete problem: A problem that cannot be solved by an algorithm in polynomial time.

Parametric faults: Non-logical faults.

Permanent faults: Faults which do not change over time. Also called solid faults.

Physical fault: A physical defect in a circuit.

Preset experiment: An experiment in which the entire test set is specified in advance. Virtually all fault detection experiments are preset [Lala 85:51].

Primary inputs: "The externally accessible input pins of a circuit through which we can inject logical values into the circuit" [KirkI 88:48].

Primary outputs: "The externally accessible output pins of the circuit through which we can observe logical values from the circuit" [KirkI 88:48].

Primitive procedures: Predefined procedures in an implementation of the Scheme programming language which may be evaluated at any time within a Scheme environment.

Reconvergent fanout: A condition such that at least two fanout branches from the same fanout form paths which come together at inputs to one or more gates between the fanout and a primary output.

Redundant circuit: One in which there exists a line which could be cut without changing the function implemented by the circuit [Nagle 75:497].

Redundant faults: Faults that cannot be detected by a diagnostic experiment, because a circuit performs correctly even though they exist. These faults are named as such because they occur in the lines of a redundant circuit which could be cut without changing the function of the circuit.

Sequential circuits: Circuits with some form of memory, i.e., circuits whose outputs depend on previous values of the inputs and/or the outputs as well as current values of the inputs.

Signal assignment statement: A VHDL statement of the form

```
target <= waveform
```

where **target** is a signal which receives the value of **waveform**. **Waveform** is composed of a collection of signals and operators which combine to form a value which is then assigned to **target**. [IEEE 88.8-3]

Solid faults: Permanent faults.

Structural VHDL: The view of VHDL which describes a design entity by a collection of lower-level design entities, e.g., a combinational logic circuit can be described by a collection of design entities representing the different types of gates that compose the circuit. Each of the gates would have their own associated entity declaration and architecture body.

Stuck-at faults: Logical faults in which a line is constantly at a high voltage level (*stuck-at-1*) or at a low level (*stuck-at-0*) [Al-Ar 87b:360].

Stuck-at-0 fault: A fault in which a line is constantly at a low voltage level resulting in a constant "0" on the line (positive logic).

Stuck-at-1 fault: A fault in which a line is constantly at a high voltage level resulting in a constant "1" on the line (positive logic).

Stuck-on fault: A fault, unique to the MOS technology used in integrated circuits, in which a transistor is stuck in a closed or on state. MOS transistors are switches which are either open (off) or closed (on).

Stuck-open fault: A fault, unique to the MOS technology used in integrated circuits, which makes a combinational circuit element into a sequential element. Stuck-open faults are caused by a transistor getting disconnected from the circuit due to being stuck in an open state. MOS transistors are switches which are either open (off) or closed (on). [Al-Ar 87b:360]

Temporary fault: A fault which changes with time. The two types of temporary faults are transient and intermittent faults [Lala 85:20].

Test: The application of a single test vector to a circuit.

Test set: A sequence of test vectors applied to a circuit in a diagnostic experiment. The test vectors may be applied either in a predesignated sequence or in an arbitrary sequence.

Test vector: An array of signals simultaneously applied to all inputs of a circuit during a test.

Transient faults: Temporary faults that occur arbitrarily.

Transformation: Typically involves the conversion from one view of a circuit to another. For example, a transistor-level view of the circuit may be transformed to a gate-level description. A diagnostic system then operates on the new view; tests generated for the new view detect faults which occur in the original representation of the circuit.

Value justification: A process used in effect-cause analysis in which an attempt is made to justify the states of internal nodes in a circuit depending on how the circuit responded to a given test vector [Abram 80:454].

VHDL: The VHSIC Hardware Description Language, an IEEE standard hardware description language used for specifying and simulating digital systems.

VLSI: Very Large Scale Integration. Devices which contain greater than 5000 gates [Johns 87:72].

Appendix B. Fundamentals of Boolean Algebra

Definitions

An algebra is characterized by three components:

1. A set, called a *carrier*,
2. *Operations* defined on the carrier, and
3. Distinct members of the carrier which are called *constants* of the algebra.[Stana 77:301]

In addition to these components, an algebra has associated *axioms*. A *closed algebraic system* is governed by the Law of Substitution which states that two expressions are said to be equal if one can be replaced by the other [Nagle 75:55].

A *Boolean algebra* is a closed algebraic system denoted by the quintuple

$$\langle \mathbf{B}, +, \cdot, 0, 1 \rangle \quad (\text{B.1})$$

where

- \mathbf{B} is the carrier of the algebra,
- $+$ and \cdot are binary operations defined on \mathbf{B} , and
- 0 and 1 are the constants of \mathbf{B} .

The operator \cdot is called AND. An expression of the form $a \cdot b$ is called a *conjunction*.

The operator $+$ is called OR. An expression of the form $a + b$ is called a *disjunction*.

The $*$ symbol is often used in lieu of the \cdot symbol. Additionally, $a \cdot b$ may be replaced by the juxtaposition ab for simplicity.

Axioms

A Boolean algebra is based on a set of axioms known as Huntington's Postulates [Hunti 04].

These axioms are:

1. **Commutative Laws.** For all $a, b \in \mathbf{B}$,

$$a + b = b + a \quad (\text{B.2})$$

$$a \cdot b = b \cdot a. \quad (\text{B.3})$$

2. **Distributive Laws.** For all $a, b, c \in \mathbf{B}$,

$$a + (b \cdot c) = (a + b) \cdot (a + c) \quad (\text{B.4})$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c). \quad (\text{B.5})$$

3. **Identities.** For all $a \in \mathbf{B}$,

$$0 + a = a \quad (\text{B.6})$$

$$1 \cdot a = a. \quad (\text{B.7})$$

0 is the identity for the + operator. 1 is the identity for the · operator.

4. **Complements.** For every $a \in \mathbf{B}$, there exists an $a' \in \mathbf{B}$ such that

$$a + a' = 1 \quad (\text{B.8})$$

$$a \cdot a' = 0. \quad (\text{B.9})$$

The " " symbol denotes *complementation*.

Boolean algebras are governed by the *principle of duality* by which a given valid expression has an associated valid dual expression. The dual of an expression is found by interchanging all + and · operators and interchanging identity elements 0 and 1. Note that each of the preceding postulates has two expressions; these expressions are duals of each other.

The Inclusion Relation

A relation, \leq , is defined as follows. For all $a, b \in \mathbf{B}$

$$a \leq b \Leftrightarrow a \cdot b' = 0 \quad (\text{B.10})$$

[Rudea 74:8]

The relation \leq is called the *inclusion relation*.

Theorems

Theorems which can be proven from the axioms and the definition of the inclusion relation are:

1. **Associativity.** For all $a, b, c \in \mathbf{B}$,

$$a + (b + c) = (a + b) + c \quad (\text{B.11})$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c. \quad (\text{B.12})$$

2. **Idempotence.** For all $a \in \mathbf{B}$,

$$a + a = a \quad (\text{B.13})$$

$$a \cdot a = a. \quad (\text{B.14})$$

3. **Boundedness.** For all $a \in \mathbf{B}$,

$$a + 1 = 1 \quad (\text{B.15})$$

$$a \cdot 0 = 0. \quad (\text{B.16})$$

4. **Absorption.** For all $a, b \in \mathbf{B}$,

$$a + (a \cdot b) = a \quad (\text{B.17})$$

$$a \cdot (a + b) = a. \quad (\text{B.18})$$

5. **Involution.** For all $a \in \mathbf{B}$,

$$(a')' = a. \quad (\text{B.19})$$

6. **DeMorgan's Laws.** For all $a, b \in \mathbf{B}$,

$$(a + b)' = a' \cdot b' \quad (\text{B.20})$$

$$(a \cdot b)' = a' + b'. \quad (\text{B.21})$$

7. For all $a, b \in \mathbf{B}$,

$$a + a' \cdot b = a + b \quad (\text{B.22})$$

$$a \cdot (a' + b) = a \cdot b. \quad (\text{B.23})$$

8. **Consensus.** For all $a, b, c \in \mathbf{B}$,

$$a \cdot b + a' \cdot c + b \cdot c = a \cdot b + a' \cdot c \quad (\text{B.24})$$

$$(a + b) \cdot (a' + c) \cdot (b + c) = (a + b) \cdot (a' + c). \quad (\text{B.25})$$

9. **Interchange.** For all $a, b, c \in \mathbf{B}$,

$$(a \cdot b) + (a' \cdot c) = (a + c) \cdot (a' + b) \quad (\text{B.26})$$

$$(a + b) \cdot (a' + c) = (a \cdot z) + (a' \cdot b). \quad (\text{B.27})$$

10. For all $a, b \in \mathbf{B}$,

$$a \leq a + b \quad (\text{B.28})$$

$$a \cdot b \leq a. \quad (\text{B.29})$$

[Johns 87, Lipsch 76, Nagle 75]

Properties

General properties of Boolean algebras which can be proven from the postulates and theorems are:

1.

$$a = b \Leftrightarrow a' \cdot b + a \cdot b' = 0 \quad (\text{B.30})$$

$$a = b \Leftrightarrow a' \cdot b' + a \cdot b = 1. \quad (\text{B.31})$$

$(a' \cdot b + a \cdot b')$ is the exclusive-OR of a and b and is denoted by either $(a \oplus b)$ or a XOR b .
 $(a' \cdot b' + a \cdot b)$ is the exclusive-NOR of a and b and is denoted be either $(a \odot b)$ or a XNOR b .

2.

$$a = 0 \& b = 0 \Leftrightarrow a + b = 0 \quad (\text{B.32})$$

$$a = 1 \& b = 1 \Leftrightarrow a \cdot b = 1. \quad (\text{B.33})$$

Literals, Terms, and Formulas

A *literal* is a variable or complemented variable such as a, b, a', b' . A *term* is a 1, a literal, or a conjunction of two or more literals in which no two literals involve the same variable. Examples of terms include ab' , ac , and abc' . An *alterm* is a 0, a literal, or a disjunction of literals in which no two literals involve the same variable. Examples include $(a + b)$, $(a + c')$, and $(a + b + c')$.

[Brown 88a:2.1-1][Lipsc 76:225]

The set of Boolean *formulas* on n symbols x_1, \dots, x_n is defined by the following:

1. The elements of \mathbf{B} are Boolean formulas, and
2. The symbols x_1, \dots, x_n are Boolean formulas, and
3. If f and g are Boolean formulas, then so are
 - (a) $f + g$,
 - (b) $f \cdot g$,
 - (c) f' , and

4. A string is a Boolean formula if and only if it is formed by a finite number of applications of the first three rules.

Examples of formulas include $x, x', x + y, (x \cdot (y + z))' + w$.

A *sum-of-products* formula is 0, a single term, or a disjunction of terms. A *product-of-sums* formula is 1, a single alterm, or a conjunction of alterms. [Brown 88a:2.1-1]

Functions

An n -variable Boolean *function*, $f : B^n \rightarrow B$, is the mapping associated with an n -variable Boolean formula. Rudeanu, in his work on Boolean functions and equations, gives an informal definition of a Boolean function:

Roughly speaking, a *Boolean function* (also called *Boolean polynomial* by certain authors) is a function with arguments and values in a Boolean algebra B , such that f can be obtained from variables and constants of B by superpositions of the basic operations $+, \cdot, ',$ and $'$ of B . [Rudea 74:16]

Rudeanu makes a clear distinction between Boolean functions in the general case, and the special case of Boolean functions involving no constants except 0 and 1 which he calls simple Boolean functions [Rudea 74:xvi]. He states:

In the particular case of the two-element Boolean algebra $B_2 = \{0, 1\}$, every function $f : B_2^n \rightarrow B_2$ is a simple Boolean function and will be termed a *truth function* (also called a "switching function" or "Boolean function" by switching theorists ...) [Rudea 74:xvi]

The switching theorist point of view is taken in this work; however, all axioms, properties, and theorems discussed in this report hold for Boolean functions in the general case.

Boolean functions may be constructed as follows:

1. For n variables, x_1, \dots, x_n , the *projection function* $f : B_2^n \rightarrow B_2$ defined by

$$f(x_1, \dots, x_n) = x_i \quad \forall (x_1, \dots, x_n) \in B_2^n, \quad i \in \{1 \dots n\}, \quad (\text{B.34})$$

is an n -variable Boolean function.

2. If $g, h : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$ are n -variable Boolean functions, then the functions $g + h, gh$, and g' defined by

(a)

$$(g + h)(x_1, \dots, x_n) = g(x_1, \dots, x_n) + h(x_1, \dots, x_n) \quad (\text{B.35})$$

(b)

$$gh(x_1, \dots, x_n) = g(x_1, \dots, x_n) \cdot h(x_1, \dots, x_n) \quad (\text{B.36})$$

(c)

$$g'(x_1, \dots, x_n) = (g(x_1, \dots, x_n))' \quad (\text{B.37})$$

$\forall (x_1, \dots, x_n) \in \mathbf{B}_2^n$, are also n -variable Boolean functions.

3. A function is a Boolean function if and only if it is formed by a finite number of applications of the first two rules. [Rudea 74:17]

Every n -variable Boolean formula maps into a corresponding n -variable Boolean function. A function, $f : \mathbf{B}^n \rightarrow \mathbf{B}$, is a Boolean function if and only if it can be represented by a Boolean formula. Moreover, a Boolean function may have any number of corresponding formulas. Formulas that represent the same function are called *equivalent formulas*. A *function table* or *truth table* is often used to specify a function.

Example B.1:

Given the two-element Boolean algebra, $\mathbf{B} = \{0, 1\}$, a truth table for the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ corresponding to the Boolean formula $xyz + x'z' + y'z'$ is given by Table B.1:

x	y	z	$f(x, y, z)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table B.1. Truth Table for Example B.1

□

Boolean Expansion Theorem

The most important functional theorem is the *Boolean Expansion Theorem*. It is stated as follows:

If f is an n -variable Boolean function, then f has the expansions

$$f(x_1, x_2, \dots, x_n) = x'_1 f(0, x_2, \dots, x_n) + x_1 f(1, x_2, \dots, x_n) \quad (\text{B.38})$$

$$f(x_1, x_2, \dots, x_n) = [x'_1 + f(1, x_2, \dots, x_n)][x_1 + f(0, x_2, \dots, x_n)]. \quad (\text{B.39})$$

[Boole 54]

Extended Verification Theorem

Another important theorem in Boolean algebra is the *Extended Verification Theorem*. It is stated:

Let $f, g : \mathbf{B}^n \rightarrow \mathbf{B}$ be Boolean functions, and assume that the equation $f(X) = 0$ is consistent.

Then the following statements are equivalent:

1. $f(X) = 0 \Rightarrow g(X) = 0$,
2. $g(X) \leq f(X) \quad \forall X \in \mathbf{B}^n$,
3. $g(X) \leq f(X) \quad \forall X \in \{0, 1\}^n$

[Rudea 74:100]

Canonical Forms

It is often desirable to use a restricted class of formula in which any Boolean function has only one corresponding formula. Formulas in such classes are called *canonical forms*. Canonical Boolean forms include the *minterm canonical form*, the *maxterm canonical form*, and the *Blake canonical form*.

Minterm Canonical Form. A *minterm* is a term in a formula of n variables which contains all variables of the formula either in complemented or uncomplemented form. A formula in *minterm canonical form* is a sum-of-products formula in which all of the terms are minterms. A minterm canonical form is also called a *canonical sum-of-products form* or *full disjunctive normal form* [Lipsc 76:225][Nagle 75:84].

Example B.2:

Given the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ from Example B.1, the following formula in minterm canonical form represents the same function f :

$$xyz + x'yz' + x'y'z' + xy'z'. \quad (\text{B.40})$$

□

Often, a shorthand notation is used to represent a minterm. This form is m_i , where i is the decimal integer of the binary code for the minterm. The shorthand notation for three-variable minterms is given in Table B.2.

Term	Binary Code	Shorthand Notation
$x'y'z'$	0 0 0	m_0
$x'y'z$	0 0 1	m_1
$x'yz'$	0 1 0	m_2
$x'yz$	0 1 1	m_3
$xy'z'$	1 0 0	m_4
$xy'z$	1 0 1	m_5
xyz'	1 1 0	m_6
xyz	1 1 1	m_7

Table B.2. Shorthand Notation for Minterms

Using this notation, the formula in Example B.2 can be written as $f(x, y, z) = m_0 + m_2 + m_4 + m_7$. This notation can be shortened further to *minterm list form*. The function $f(x, y, z)$ is expressed in minterm list form as $f(x, y, z) = \sum m(0, 2, 4, 7)$. [Nagle 75:85]

Maxterm Canonical Form. A *maxterm* is an alterm in a formula of n variables which contains all variables of the formula either in complemented or uncomplemented form. A formula in *maxterm canonical form* is a product-of-sums formula in which all of the alterms are maxterms. A maxterm canonical form is also called a *canonical product-of-sums form* or *full conjunctive normal form* [Lipsc 76:225][Nagle 75:84]. Hence, the maxterm canonical form is analogous to the minterm canonical form where the formula is expressed in product-of-sums form rather than sum-of-products form and terms are replaced by alterms.

Example B.3:

Given the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ from Example B.1, the following formula in product-of-sums form represents the same function f :

$$(x + z')(x' + y + z')(x' + y' + z) \quad (\text{B.41})$$

This formula can be transformed to the following formula in maxterm canonical form:

$$(x + y + z')(x + y' + z')(x' + y + z')(x' + y' + z) \quad (\text{B.42})$$

□

As with minterms, a shorthand notation is used to represent maxterms. This form is M_i , where i is the decimal integer of the binary code for the maxterm. The shorthand notation for three-variable maxterms is given in Table B.3. Using this notation, the formula in Example B.3 can be written as $f(x, y, z) = M_1 M_3 M_5 M_6$. This notation can be shortened further to *maxterm list form*. The function $f(x, y, z)$ is expressed in maxterm list form as $f(x, y, z) = \prod M(1, 3, 5, 6)$. [Nagle 75:88]

Blake Canonical Form. A term p is called an *implicant* of a Boolean function f if $p \leq f$. When a function f is expressed in sum-of-products form, all terms in the form are implicants of f . A *prime implicant* of a Boolean function f is an implicant of f such that it is no longer an implicant if any of its literals is removed [Quine 52]. Boolean axioms and theorems such as consensus and

Alterm	Binary Code	Shorthand Notation
$x + y + z$	0 0 0	M_0
$x + y + z'$	0 0 1	M_1
$x + y' + z$	0 1 0	M_2
$x + y' + z'$	0 1 1	M_3
$x' + y + z$	1 0 0	M_4
$x' + y + z'$	1 0 1	M_5
$x' + y' + z$	1 1 0	M_6
$x' + y' + z'$	1 1 1	M_7

Table B.3. Shorthand Notation for Maxterms

absorption are used to reduce a Boolean formula for a function to a form which consists of the prime implicants of the function. An application is minimization, one approach to which is to reduce a Boolean formula to an equivalent formula which includes the smallest number of prime implicants that still represent the same function. The impetus for minimization is to represent a Boolean function by a formula that can be implemented in hardware with the smallest number of components. See [Nagle 75, Quine 52, Quine 55] for discussions of Boolean minimization. A *prime implicate* is the analog of a prime implicant for the product-of-sums form.

Example B.4.

The only term in the n -variable Boolean formula f given by

$$xyz + x'yz' + x'y'z' + xy'z' \quad (\text{B.43})$$

that is a prime implicant of f is xyz . The formula may be transformed to an equivalent formula consisting of only prime implicants by application of Boolean axioms and theorems. An equivalent formula which consists only of prime implicants is:

$$xyz + y'z' + x'z'. \quad (\text{B.44})$$

□

In the process of reducing a given formula to prime implicants, *superfluous* terms are often generated. A term p is superfluous in a sum-of-products formula, $p + q$, if $p + q$ is equivalent

to the formula q [Quine 52:522]. A literal of a term in a sum-of-products formula is *superfluous* if it can be removed without changing the formula to a non-equivalent formula. Quine called a "formula *irredundant* if it has no superfluous clauses and none of its clauses has superfluous literals [Quine 52:523]."

Another application for the prime implicants of a formula is for *Boolean inference*, also called *Boolean reasoning*. Boolean inference is "the extraction of conclusions from a collection of Boolean data" [Brown 88a:2.0-2]. The basis for Boolean inference is the *Blake canonical form*. The Blake canonical form, denoted $BCF(f)$, of a function f is the disjunction of all of the prime implicants of f . The Blake canonical form is a complete and simplified representation of all possible conclusions that can be inferred from a Boolean equation. Methods for generating $BCF(f)$ are by the *exhaustion of implicants*, *iterated consensus*, and *multiplication*. Blake invented the methods of iterated consensus and multiplication [Blake 37]. Iterated consensus is discussed in [Quine 52]; the multiplication method is found in [Samso 54].

Example B.5.

The n -variable Boolean function defined in Table B.1 and represented by the formula

$$xyz + y'z' + x'z' \quad (B.45)$$

is in Blake canonical form because the formula consists of all of the prime implicants of the function.

□

Reduction

Any system of Boolean equations can be reduced to a single Boolean equation of the form $f(\underline{x}) = g(\underline{x})$ where $g(\underline{x})$ is any preassigned Boolean function [Rudea 74:116-117]. In particular, we

may choose $g(\underline{x})$ to be 0 or 1. (The notation \underline{x} denotes the vector (x_1, x_2, \dots, x_n) .) The form $f(\underline{x}) = 0$ is derived in the following manner. A system

$$\begin{aligned} g_1(\underline{x}) &= h_1(\underline{x}) \\ g_2(\underline{x}) &= h_2(\underline{x}) \\ &\vdots \\ g_n(\underline{x}) &= h_n(\underline{x}) \end{aligned} \tag{B.46}$$

of Boolean equations can be transformed, using property (B.30), into the equivalent system

$$\begin{aligned} g_1(\underline{x}) \oplus h_1(\underline{x}) &= 0 \\ g_2(\underline{x}) \oplus h_2(\underline{x}) &= 0 \\ &\vdots \\ g_n(\underline{x}) \oplus h_n(\underline{x}) &= 0. \end{aligned} \tag{B.47}$$

This system of equations can then be transformed into a single Boolean equation by property (B.32). Since all of the equations must be simultaneously true, they are “&’ed” together as in equation (B.32). However, the “&” symbol is dropped for notational simplicity. The resulting single Boolean equation is

$$f(\underline{x}) = 0 \tag{B.48}$$

where f is defined by

$$f = \sum_{i=1}^n g_i \oplus h_i, \tag{B.49}$$

i.e.,

$$f = \sum_{i=1}^n (g'_i h_i + g_i h'_i). \tag{B.50}$$

The $p(\underline{x}) = 1$ form of a system of equations is similarly derived. The system of equations (B.46) can be transformed into an equivalent system using the property shown by equation (B.31):

$$\begin{aligned} g_1(\underline{x}) \odot h_1(\underline{x}) &= 1 \\ g_2(\underline{x}) \odot h_2(\underline{x}) &= 1 \\ &\vdots \\ g_n(\underline{x}) \odot h_n(\underline{x}) &= 1. \end{aligned} \quad (\text{B.51})$$

This system of equations is transformed into a single Boolean equation by equation (B.33). Again, the “&” symbol is dropped for notational simplicity. The resulting single Boolean equation is

$$p(\underline{x}) = 1 \quad (\text{B.52})$$

where p is defined by

$$p = \prod_{i=1}^n (g_i \odot h_i), \quad (\text{B.53})$$

i.e.,

$$p = \prod_{i=1}^n (g'_i h'_i + g_i h_i). \quad (\text{B.54})$$

The utility of the choice of the $f(\underline{x}) = 0$ form versus the $p(\underline{x}) = 1$ form is dependent on the application [Rudea 74:52]. Conversion between the two forms is done by complementation of both sides of the equality, i.e.,

$$f'(\underline{x}) = 0 \Leftrightarrow f(\underline{x}) = 1 \quad (\text{B.55})$$

and

$$p'(\underline{x}) = 1 \Leftrightarrow p(\underline{x}) = 0. \quad (\text{B.56})$$

Eliminants

The Conjunctive Eliminant. For an n -variable Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ with variables x_1, \dots, x_n and a subset $\{x_1, x_2\}$ of the variables, the *conjunctive eliminant* of the function with respect to $\{x_1, x_2\}$ is defined as:

$$ECON(f, \{x_1, x_2\}) = \prod_{(x_1, x_2) \in \{0,1\}^2} f(x_1, x_2, x_3, \dots, x_n). \quad (\text{B.57})$$

[Brown 88a:3.8-1]

Although a specific subset of the variables was used in the above definition, the conjunctive eliminant of a function may be found with respect to an arbitrary subset of the variables in the function.

Example B.6:

The conjunctive eliminant of a function $f(x, y, z)$ with respect to z is given by

$$ECON(f(x, y, z), z) = f(x, y, 0)f(x, y, 1). \quad (\text{B.58})$$

□

Brown has shown that the conjunctive eliminant of a function in Blake canonical form with respect to a given variable is the sum of terms in the form which do not involve the variable [Brown 88a:3.8-2]. Formally,

$$ECON(f, \{y\}) = \sum (\text{terms of } BCF(f) \text{ which do not have a literal } y \text{ or } y'). \quad (\text{B.59})$$

The resulting formula is in Blake canonical form.

The Disjunctive Eliminant. For an n -variable Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ with variables x_1, \dots, x_n and a subset $\{x_1, x_2\}$ of the variables, the *disjunctive eliminant* of the function with respect to $\{x_1, x_2\}$ is defined as:

$$EDIS(f, \{x_1, x_2\}) = \sum_{(x_1, x_2) \in \{0,1\}^2} f(x_1, x_2, x_3, \dots, x_n). \quad (\text{B.60})$$

[Brown 88a:3.8-1]

As in the conjunctive eliminant, the disjunctive eliminant of a function may be found with respect to an arbitrary subset of the variables in the function.

Example B.7:

The disjunctive eliminant of the function $f(x, y, z)$ with respect to z is

$$EDIS(f(x, y, z), z) = f(x, y, 0) + f(x, y, 1). \quad (\text{B.61})$$

□

A simple method for deriving the disjunctive eliminant of a Boolean function f is by transforming the formula that represents the function to any equivalent sum-of-products form and then replacing the literals of the variables to be eliminated, whether in complemented or uncomplemented form, by 1 [Mitch 83].

Elimination

Given a Boolean equation, it is possible to determine constraints on certain variables given the absence of information with respect to the other variables using a process called *elimination*. Equations deduced as the result of elimination are called *resultants of elimination*.

Using the definition of the conjunctive eliminant, a variable may be eliminated from an equation to form a new equation:

$$f(\underline{x}) = 0 \Rightarrow ECON(f, \{x_i\}) = 0 \quad (\text{B.62})$$

The equation $ECON(f, \{x_i\}) = 0$ is called the resultant of elimination of x_i from equation $f(\underline{x}) = 0$.

Using the definition of the disjunctive eliminant, a variable may be eliminated from an equation to form a new equation:

$$p(\underline{x}) = 1 \Rightarrow EDIS(p, \{x_i\}) = 1 \quad (\text{B.63})$$

The equation $EDIS(f, \{x_i\}) = 1$ is called the resultant of elimination of x_i from equation $p(\underline{x}) = 1$.

Solutions of Boolean Equations

A solution of the equation $f(\underline{x}) = 0$ is a vector $\underline{a} \in \mathbf{B}^n$ such that $f(\underline{a}) = 0$ is an identity. In general, it is inconvenient to determine solutions of the $f(x, y, z) = 0$ form of an equation. A simple method to find a solution to an equation is first to convert the equation to the equivalent $p(x, y, z) = 1$ form as in equation (B.55), and then express p in minterm canonical form. Solutions are found by inspection of the minterms of $p(x, y, z)$.

Example B.8:

Given the equation

$$xyz' + x'z + y'z = 0, \quad (\text{B.64})$$

the $f(\underline{x}) = 1$ form of this equation is

$$xyz + x'z' + y'z' = 1. \quad (\text{B.65})$$

The minterm canonical form of the left-hand side of this equation is used to form a new equation

$$xyz + x'y'z' + x'y'z + xy'z' = 1. \quad (\text{B.66})$$

By inspection, solutions of the equation are

$$(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1). \quad (\text{B.67})$$

□

An equation typically will have several solutions. Constant vectors, \underline{a} and \underline{b} , are called *equivalent* with respect to f if $f(\underline{a}) = f(\underline{b})$. Two equations are called “*equivalent* if they have the same set of solutions” [Rudea 74:50].

Comparison of Functions

Given two n -variable Boolean functions f and g , a function h can be constructed which shows all circumstances in which functions f and g are different. h is defined in the following way:

$$f \oplus g = h \quad (\text{B.68})$$

Minterms of h define the differences between f and g .

Example B.9:

Given the equations $f(x, y) = x$ and $g(x, y) = y$, $h(x, y)$ is found as follows:

$$\begin{aligned} h(x, y) &= f(x, y) \oplus g(x, y) \\ &= x \oplus y. \end{aligned} \quad (\text{B.69})$$

Minterms of $h(x, y)$ are xy' and $x'y$. The results are summarized in Table B.4. \square

$x \ y$	$f(x, y)$	$g(x, y)$	$h(x, y)$
0 0	0	0	0
0 1	1	0	1
1 0	0	1	1
1 1	1	1	0

Table B.4. Results of Example B.9

Appendix C. Definition of VHDL Subset

This appendix defines the subset of VHDL used in the combinational circuit diagnostic system.

All definitions are taken from the IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987 [IEEE 88]. Productions are strict extractions from the Reference Manual. However, in many cases where VHDL allows options, the options are restricted to conform to the requirements of the diagnostic system. Each production is listed with the appropriate page number from the Reference Manual.

The intent of this subset is to describe a combinational circuit by a single VHDL architecture and a gate by a single signal assignment statement. Thus, in addition to the definitions that ensue the following restrictions also apply:

- A 3-Input AND gate is modeled as follows:

Output \leq A and B and C;

- A 3-Input OR gate is modeled as follows:

Output \leq A or B or C;

- A 2-Input XOR gate is modeled as follows:

Output \leq A xor B;

- A NOT gate may be modeled as:

Output \leq not A;

or:

Output \leq not(A);

- A 3-Input NAND gate is modeled as follows:

Output \leq not(A and B and C);

- A 3-Input NOR gate is modeled as follows:

Output \leq not(A or B or C);

Gates with more inputs are simple extensions of the preceding examples. More complex expressions, although allowable by VHDL, may not be processed properly by the diagnostic system. The *nand* and *nor* operators have been excluded due to restrictions on associativity [IEEE 88:7-2].

BNF Notation

Definitions are in Backus-Naur Form (BNF) notation, a common way to define productions in a grammar. A description of this notation, as used in these definitions follows:

- The symbol “::=” denotes “is defined as.”
- A vertical bar, |, separates alternative items.
- Square brackets, [], enclose optional items.
- Braces, { }, enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent [IEEE 88:Introduction]:

term ::= factor {operator factor}

term ::= factor | term operator factor

Entity Declarations

- The "entity declaration defines the interface between a given design entity and the environment in which it is used." [IEEE 88:1-1]
- "An entity declaration can potentially represent a class of design entities, each with the same interface." [IEEE 88:1-1]

```
-- (page 1-1)
entity_declarator ::= 
    entity identifier is
        entity_header
    end [entity_simple_name];
```

- The entity_simple_name exactly repeats the identifier [IEEE 88:1-2].

```
-- (page 1-2)
entity_header ::= 
    [formal_port_clause]

-- (page 1-2)
port_clause ::= 
    port(port_list);

-- (page 1-3)
port_list ::= 
    port_interface_list
```

- The port_interface_list is nothing more than an interface_list.

Architecture Declarations

- The “architecture defines the body of a design entity. It specifies the relationships between the inputs and outputs of a design entity, and may be expressed in terms of structure, dataflow, or behavior.” [IEEE 88:1-6]

```
-- (page 1-6)
architecture_body ::= 
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [architecture_simple_name];
```

- “The identifier defines the simple name of the architecture body; this simple name distinguishes architecture bodies associated with the same entity declaration.” [IEEE 88:1-6]
- “The entity_name identifies the name of the entity declaration that defines the interface of this design entity.” [IEEE 88:1-6]
- The architecture_simple_name exactly repeats the identifier. [IEEE 88:1-6]

```
-- (page 1-7)
architecture_declarative_part ::= 
    {block_declarative_item}

-- (page 1-7)
block_declarative_item ::= 
    signal_declaration

-- (page 4-5)
signal_declaration ::= 
    signal identifier_list : subtype_indication;

-- (page 1-7)
architecture_statement_part ::= 
    {concurrent_statement}
```

Concurrent Statements

```
-- (page 9-1)
concurrent_statement ::= 
    concurrent_signal_assignment_statement

-- (page 9-6)
concurrent_signal_assignment_statement ::= 
    conditional_signal_assignment_statement

-- (page 9-9)
conditional_signal_assignment_statement ::= 
    target <= conditional_waveforms;

-- (page 9-9)
conditional_waveforms ::= 
    waveform
```

- Note: The concurrent_statement simply transforms to:

```
concurrent_statement ::= signal_assignment_statement
signal_assignment_statement ::= target <= waveform;
```

```
-- (page 8-3)
target ::= 
    name

-- (page 6-1)
name ::= 
    simple_name

-- (page 6-2)
simple_name ::= 
    identifier

-- (page 8-3)
waveform ::= 
    waveform_element

-- (page 8-4)
waveform_element ::= 
    value_expression [after time_expression]
```

- The value_expression is nothing more than an expression.

Expressions

```
-- (page 7-1)
expression ::= 
    relation {and relation} |
    relation {or relation} |
    relation {xor relation}

-- (page 7-1)
relation ::= 
    simple_expression

-- (page 7-1)
simple_expression ::= 
    term

-- (page 7-1)
term ::= 
    factor

-- (page 7-1)
factor ::= 
    primary |
    not primary

-- (page 7-1)
primary ::= 
    name |
    (expression)
```

Port Interface List/Declarations

```
-- (page 4-11)
interface_list ::= 
    interface_element{;interface_element}

-- (page 4-11)
interface_element ::= 
    interface_declarator

-- (page 4-8)
interface_declarator ::= 
    interface_signal_declarator

-- (page 4-9)
interface_signal_declarator ::= 
    [signal] identifier_list : [mode] subtype_indication

-- (page 4-9)
mode ::= 
    in | out

-- (page 4-2)
subtype_indication ::= 
    type_mark

-- (page 4-2)
type_mark ::= 
    type_name

type_name ::= 
    bit
```

Identifiers

```
-- (page 3-13)
identifier_list ::= identifier{,identifier}

-- (page 13-4)
identifier ::= letter{[underline]letter_or_digit}

-- (page 13-4)
letter_or_digit ::= letter | digit

-- (page 13-4)
letter ::= upper_case_letter | lower_case_letter

-- (page 13-1)
digit ::= 0|1|2|3|4|5|6|7|8|9

-- (page 13-1)
upper_case_letter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

-- (page 13-2)
lower_case_letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

Appendix D. Diagnostic System Code

This appendix is the fully-commented source code for all procedures of the diagnostic system.

The system is composed of procedures in eleven files. The composition of the files parallels the system architecture described in Chapter 5, although some procedures are used throughout different modules of the system. Each file includes a header which describes in general terms the operations performed by the procedures in the file. Each procedure includes comments which describe its operation.

The following is a short synopsis of each file of the implementation of the diagnostic system.

- **diagnose.s:** Contains the main procedure which calls procedures which comprise the modules of the diagnostic system. Includes procedures used to interface to the MS DOS operating system.
- **inp-mod.s:** Includes the primary procedures of the input module.
- **tokenize.s:** Contains the procedures used to read an input file and form a list of tokens.
- **prefixer.s:** All procedures required to convert the list of tokens to the intermediate format are included in this file.
- **tok-vhdl.s:** Procedures necessary to handle VHDL-specific input data are included in this file.
- **eqn-gen.s:** Contains many of the procedures associated with the equation generation module of the diagnostic system.
- **eqn-gen.a.s:** This file is a continuation of the procedures of file eqn-gen.s. It includes the remaining procedures associated with the equation generation module of the diagnostic system.
- **tester.s:** Contains procedures which implement the tester module of the diagnostic system.
- **interp.s:** Contains procedures which implement the interpretation module of the diagnostic system.
- **interp-a.s:** This file is a continuation of the procedures of file interp.s. It includes the remaining procedures which form the interpretation module.
- **boolean.s:** This file is a compendium of procedures which implement many of the Boolean operations outlined in Appendix B, Fundamentals of Boolean Algebra. The data structure used by these procedures is the sum-of-products list notation described in Chapter 6, Implementation of the Diagnostic System. Procedures in this file were implemented by Dr. Frank M. Brown, Professor of Electrical Engineering, Air Force Institute of Technology [Brown 88b].

```

;;;;;;
;;          Filename: diagnose.s
;;
;;
;; This is the main file which calls all of the other procedures
;; in the diagnostic system. The main procedure in this file is a
;; shell from which all primary modules in the system are accessed.
;; In addition, an operating system interface is provided which
;; provides for the automatic creation of a session transcript.
;; The system allows the user to process a circuit description and
;; save the derived data structures in a file prior to the
;; execution of the diagnostic input-output experiment. Then, when
;; the circuit has been constructed, this data can be read from the
;; file to perform the diagnostic experiment. The mechanisms for
;; storing and retrieving the data are provided by procedures in
;; this file.
;;
;;
;;
;; Requires the files: tokenize.fsl, prefixer.fsl, inp-mod.fsl,
;; eqn-gen.fsl, eqn-gena.fsl, boolean.fsl,
;; tester.fsl, interp.fsl, interp-a.fsl
;; tok-vhdl.fsl
;;;;;;
;;
;; (DIAGNOSE)
;;
;; -- DIAGNOSE prints out a welcome message, resets the diagnostic
;; system by a call to RESET-DIAGNOSTIC-SYSTEM, and manages the
;; facilities for the automatic creation of a transcript file.
;; This file is stored in file "DIAGNOSE.RUN." If a previous run
;; file exists, then it is renamed to the filename "DIAGNOSE.BAK."
;; -- DIAGNOSE then calls DIAGNOSE-1 prints a menu of system options
;; and calls the other modules of the system.

(define (diagnose)

  (newline)
  (writeln "*** The Combinational Circuit Diagnostic System - V1.0
***")

  (reset-diagnostic-system)

  (if (file-exists? "diagnose.run")
      (begin
        (if (file-exists? "diagnose.bak")
            (dos-delete "diagnose.bak")
            '())
        (dos-rename "diagnose.run" "diagnose.bak")) )

  (transcript-on "diagnose.run")

  (diagnose-1) )

```

```

;; (DIAGNOSE-1)
;;
;; -- DIAGNOSE-1 is a shell from which to call the other procedures
;;   in the diagnostic system.
;; -- It contains a menu and input-output mechanisms to prompt the
;;   user for the option that he would like to execute. After the
;;   user has made his selection, the appropriate module is called.

(define (diagnose-1)

  (newline)
  (writeln "Enter the operation that you would like to perform.")
  (writeln "Your choices are: ")
  (newline)
  (writeln "    1. Diagnose a circuit (no preprocessed input.)")
  (writeln "    2. Preprocess a circuit description.")
  (writeln "    3. Diagnose a circuit (with preprocessed input.)")
  (writeln "    4. Quit (default.)")
  (newline)
  (newline)
  (display "Enter the number of your choice --> ")

  (let ( (choice (read-line)) )

    (cond ( (equal? choice "1")
             (diagnose-a) )

          ( (equal? choice "2")
             (diagnose-preprocessor) )

          ( (equal? choice "3")
             (diagnose-b) )

          ( else
              (newline)
              (transcript-off)
              (writeln "A transcript of your session is in file"
'diagnose.run')
              (writeln "Have a nice day!")
              (newline)
              (newline) ))))

;; (RESET-DIAGNOSTIC-SYSTEM)
;;
;; -- RESET-DIAGNOSTIC-SYSTEM is called when the diagnostic system is
;;   started to reset the state of the system.

```

```

;; -- Currently, it only turns the transcript off, to avoid the
;; problem of trying to restart a transcript when one has
;; previously been started. This causes a DOS error causing loss
;; of the previous transcript file and possibly and inability to
;; leave PC-Scheme normally. DOS Chkdsk/f must then be run to
;; recover the file. If the transcript is turned off prior to
;; doing anything else in the system, no errors arise.

(define (reset-diagnostic-system)
  (transcript-off) )

;; (DIAGNOSE-A)

;; -- DIAGNOSE-A is the module which has all of the mechanisms to
;; perform a complete diagnosis of a circuit. All modules in the
;; system are called to perform a complete circuit analysis, from
;; reading the original input file, tokenizing and parsing the
;; input, creating the INTERMEDIATE-FORMAT, generating the single
;; Boolean equation, conducting the input-output experiment
;; (TESTER), and INTERPRETING the results.
;; -- If REPLY is #T, then DIAGNOSE-A calls itself, otherwise the user
;; is returned to DIAGNOSE-1 where the main menu is again printed.

(define (diagnose-a)
  (let* ( (intermediate-format (run-input-module))
          (phi                  (generate-equation
                                   intermediate-format))
          (tester-output         (tester phi))
          (reply                (interpret intermediate-format
                                   phi
                                   tester-output)) )

    (if (null? reply)
        (diagnose-1)
        (diagnose-a) )))

;; (DIAGNOSE-PREPROCESSOR)

;; -- DIAGNOSE-PREPROCESSOR is the module which takes the initial
;; circuit description from a file and processes it to generate the
;; information necessary to begin the input-output experiment.
;; This allows the user to perform time-consuming preprocessing one
;; time prior to conducting the input-output experiment.
;; Information about the circuit can be processed prior to building
;; it and then stored away in a file.
;; -- OUTPUT-FOR-FILE is a list containing the preprocessed data to be
;; stored in the file for later use.

```

```

;; -- The user is prompted for the filename in which to store the
;; preprocessed data. This filename consists of eight characters
;; or less. The system appends the extension ".ckt" onto the
;; filename, yielding a name of the form: USER-INPUT-FILENAME.ckt.
;; -- After the data is stored in the file, the file is closed and the
;; user is returned to the main menu by a call to DIAGNOSE-1.

(define (diagnose-preprocessor)

  (let* ( (intermediate-format (run-input-module))
          (phi
            (generate-equation
              intermediate-format))
          (output-for-file
            (cons intermediate-format phi)) )

    (newline)
    (writeln "Enter the name of the file to hold your preprocessed
data.")
    (writeln "The filename should be 8 characters or less with NO
extension.")
    (newline)
    (display "Enter the filename --> ")

    (let* ((output-filename (read-line))
           (filename-with-ext (string-append output-filename
                                             ".ckt")))
      (output-port
        (open-output-file filename-with-ext)))

      (write output-for-file output-port)
      (close-output-port output-port)
      (newline)
      (writeln "Your preprocessed data is now in file "
              filename-with-ext ".")
      (newline)
      (diagnose-1) )))

;; (DIAGNOSE-B)
;;
;; -- DIAGNOSE-B is the procedure that is called to perform an
;; input-output experiment using the preprocessed information
;; generated by DIAGNOSE-PREPROCESSOR.
;; -- Initially, the user is prompted for a filename of 8 characters
;; or less. The extension ".ckt" is appended to this filename. The
;; file is then opened and READ. INPUT-FROM-FILE is the data that
;; was stored in the file. A CAR and CDR are used to extract the
;; INTERMEDIATE-FORMAT and PHI, respectively. At this point, the
;; system proceeds in the same fashion as if DIAGNOSE-A were
;; executed.
;; -- If REPLY is #T, then DIAGNOSE-B calls itself. Otherwise, the
;; user is returned to the main menu by a call to DIAGNOSE-1.

```

```

(define (diagnose-b)

  (newline)
  (writeln "Enter the name of the file that holds your preprocessed
data.")
  (writeln "The filename should be 8 characters or less with NO
extension.")
  (newline)
  (display "Enter the filename --> ")

  (let* ( (input-filename (read-line))
          (filename-with-ext (string-append input-filename ".ckt"))
          (input-port (open-input-file filename-with-ext))
          (input-from-file (read input-port))
          (port-status (close-input-port input-port))

          ; the following two statements were added to print the time
          ; of day as soon as the data has been read from the file
          (dummy-variable-1 (newline))
          (dummy-variable-2 (time-of-day))

          (intermediate-format (car input-from-file))
          (phi (cdr input-from-file))
          (tester-output (tester phi))
          (reply (interpret intermediate-format
                               phi
                               tester-output)) )

    ; output the time of day
    (newline)
    (time-of-day)
    (newline)

    (if (null? reply)
        (diagnose-1)
        (diagnose-b)))))

;; (TIME-OF-DAY)
;;
;; -- TIME-OF-DAY is a utility for displaying the time of day.
;; The PC-Scheme RUNTIME primitive is used which returns the number
;; of hundredths of a second that have occurred since midnight.
;; This number must be decomposed into hundredths, seconds,
;; minutes, and finally hours.
;; -- The time is then output in the form of a 24-hour clock.

(define (time-of-day)
  (let* ( (hundredths-of-seconds (runtime))

```

```
(hundredths (modulo hundredths-of-seconds 100))
(tot-no-of-seconds (/ (- hundredths-of-seconds hundredths)
                      100))
(seconds (modulo tot-no-of-seconds 60))
(tot-no-of-minutes (/ (- tot-no-of-seconds seconds) 60))
(minutes (modulo tot-no-of-minutes 60))
(tot-no-of-hours (/ (- tot-no-of-minutes minutes) 60)) )

(display "The time is now: ")
(writeln tot-no-of-hours ":" minutes ":" seconds "." hundredths) ))
```

```

;;;;;;;;;;;;;;;;
;;           Filename: inp-mod.s
;;;
;; Requires the files: tokenize.fsl, prefixer.fsl, tok-vhdl.fsl
;;
;; This file contains the procedures which control the Input
;; Module. RUN-INPUT-MODULE is the primary procedure which calls
;; either the TOKEN->PREFIX and GET-TOKENIZED-LIST procedures, or
;; the VHDL->PREFIX and GET-VHDL-TOKENS routines.
;;
;; The list returned by either TOKEN->PREFIX or VHDL->PREFIX is in
;; the following format:
;;
;;   ( (EQ D (NOT B)) (EQ E (NOT (* A D)))
;;     (EQ F (NOT (* C D))) (EQ Z (NOT (* E F))) )
;;
;; ADD-SUFFIXES places a "--" suffix at the end of each of the
;; symbols in the prefix-format to create the INTERMEDIATE-FORMAT
;; which is returned by RUN-INPUT-MODULE. The INTERMEDIATE-FORMAT
;; is as follows:
;;
;;   ( (EQ D-- (NOT B--)) (EQ E-- (NOT (* A-- D--)))
;;     (EQ F-- (NOT (* C-- D--))) (EQ Z-- (NOT (* E-- F--))) )
;;
;; ADD-SUFFIXES and helping procedures are included in this file.
;;
;;
;;;;;;;;;;;;;;

```

```

;; (RUN-INPUT-MODULE)
;;
;; -- RUN-INPUT-MODULE is the primary procedure which calls either the
;;   TOKEN->PREFIX and GET-TOKENIZED-LIST procedures, or the
;;   VHDL->PREFIX
;;   and GET-VHDL-TOKENS routines.
;; -- The list returned by either TOKEN->PREFIX or VHDL->PREFIX is in
;;   the following format:
;;
;;   ( (EQ D (NOT B)) (EQ E (NOT (* A D))) (EQ F (NOT (* C D)))
;;     (EQ Z (NOT (* E F))) )
;;
;; -- ADD-SUFFIXES is called to place a "--" suffix at the end of each
;;   of the node symbols in the prefix-format to create the
;;   INTERMEDIATE-FORMAT which is returned.
;;
;; -- The INTERMEDIATE-FORMAT is as follows:
;;
;;   ( (EQ D-- (NOT B--)) (EQ E-- (NOT (* A-- D--)))
;;     (EQ F-- (NOT (* C-- D--))) (EQ Z-- (NOT (* E-- F--))) )

```

```

(define (run-input-module)
  (newline)
  (newline)
  (writeln "Enter the type of input file for your circuit
description.")
  (writeln "Your choices are: ")
  (newline)
  (writeln "    1. VHDL Dataflow Description")
  (writeln "    2. Boolean Equation Description")
  (newline)
  (newline)
  (display "Enter the number of your choice --> ")
  (let ( (choice (read-line)) )
    (cond ( (equal? choice "1")
              (let* ((prefix-list (vhdl->prefix (get-vhdl-tokens)))
                     (intermediate-format (add-suffixes prefix-list)))
                intermediate-format)
            ( (equal? choice "2")
              (let* ((prefix-list (token->prefix
                                         (get-tokenized-list)))
                     (intermediate-format (add-suffixes
                                         prefix-list)))
                intermediate-format)
            ( else
              (newline)
              (writeln "You must enter either 1 or 2!")
              (run-input-module) )))))

;;;;;;;SUFFIX ADDITION PROCEDURES;;;;;;;

;; (ADD-SUFFIXES prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list representing a system of equations of the
;;                 form:
;;
;;                 ( (EQ D (NOT B)) (EQ E (NOT (* A D))))
;;                   (EQ F (NOT (* C D))) (EQ Z (NOT (* E F)))) )
;;
;; -- ADD-SUFFIXES takes the initial system of equations and breaks it
;; down line by line calling ADD-SUFFIXES-1 to operate on
;; individual lines.
;; -- Thus, in the first call to ADD-SUFFIXES-1, the list
;;   (EQ D (NOT B)) is passed to ADD-SUFFIXES-1. ADD-SUFFIXES-1
;;   returns a list of the form (EQ D-- (NOT B--)). ADD-SUFFIXES
;;   reassembles the system of equations to return a list of the
;;   form:
;;

```

```

;;      ( EQ D-- (NOT B--) ) (EQ E-- (NOT (* A-- D--)))
;;      (EQ F-- (NOT (* C-- D--))) (EQ Z-- (NOT (* E-- F--))) )

(define (add-suffixes prefix-list)
  (if (null? prefix-list)
      '()
      (append (list (add-suffixes-1 (car prefix-list)))
              (add-suffixes (cdr prefix-list)))))

;; (ADD-SUFFIXES-1 inlist)
;;
;; Parameters:
;;   inlist -- a list representing a single equation of the form:
;;             (EQ D (NOT B))

;; -- ADD-SUFFIXES-1 is a helping procedure for ADD-SUFFIXES.
;; This procedure breaks up the line into the left and right halves
;; of the equality, or inclusion relations. ADD-SUFFIXES-2 is
;; called to operate on the individual components of each line.
;; -- ADD-SUFFIXES-2 adds the suffix to the formulas on the right and
;; left sides of the equality. For the given list, the formulas
;; D and (NOT B) would be passed to ADD-SUFFIXES-2 the two times it
;; is called from ADD-SUFFIXES-1.
;; -- The ASSEMBLE-LINE procedure from file PREFIXER.S is used to
;; reassemble the line into its component parts. An equation of
;; the following form is returned: (EQ D-- (NOT B--)).

(define (add-suffixes-1 inlist)
  (if (or (eq? 'EQ (car inlist))
          (eq? 'LE (car inlist))
          (eq? 'GE (car inlist)))
      (let ( (new-left-side (add-suffixes-2 (cadr inlist)))
            (new-right-side (add-suffixes-2 (caddr inlist))) )
        (assemble-line (car inlist) new-left-side new-right-side) )
      '() ))

;; (ADD-SUFFIXES-2 formula)
;;
;; -- ADD-SUFFIXES-2 is a helping procedure for ADD-SUFFIXES-1.
;; This procedure disassembles formulas of the form
;; (operator formula (operator formula formula)),
;; e.g., (* D (NOT B)).
;; -- If the input formula is actually a symbol, then GOOD-SYMBOL? is
;; called to determine whether the symbol is a valid node symbol or
;; an operator. (GOOD-SYMBOL? returns false if an operator symbol,
;; true otherwise.) If the symbol is a node, then
;; ADD-SUFFIX-TO-SYMBOL is called to append the "--" suffix to the

```

```

;;      end of the symbol. (GOOD-SYMBOL? is in file TOKENIZE.S.)
;; -- If the first element of the list is also a list, then the input
;; list is broken up and ADD-SUFFIXES-2 calls itself recursively.
;; -- Otherwise, the first element of the input formula is a symbol.
;;   If it is a node symbol, then a suffix is added to it.
;;   ADD-SUFFIXES-2 calls itself recursively to operate on the
;; remainder of the formula.

(define (add-suffixes-2 formula)
  (cond ( (null? formula)
           '() )

        ( (symbol? formula)
           (if (good-symbol? formula)
               (add-suffix-to-symbol formula)
               formula) )

        ( (list? (car formula))
           (append (list (add-suffixes-2 (car formula)))
                   (add-suffixes-2 (cdr formula)))) )

        ( else
           (if (good-symbol? (car formula))
               ; (car formula) is a node symbol
               (cons (add-suffix-to-symbol (car formula))
                     (add-suffixes-2 (cdr formula)))

               ; (car formula) is an operator
               (cons (car formula)
                     (add-suffixes-2 (cdr formula)))) )))

;; (ADD-SUFFIX-TO-SYMBOL symbol)
;;
;; Parameter:
;;   symbol - an arbitrary symbol
;;
;; -- ADD-SUFFIX-TO-SYMBOL adds the suffix "--" to the symbol.
;; -- The symbol is broken down into a string, and STRING-APPEND
;;   is used to append the "--" string to the end of the string
;;   to which the symbol was converted.
;; -- The new string is then converted back to a symbol.
;; -- If the input SYMBOL was ABC, then the output symbol would be.
;;   ABC--.

(define (add-suffix-to-symbol symbol)
  (string->symbol (string-append (symbol->string symbol) "--")))

```

```

;;;;;;;;;;;;;;;
;;          Filename: tokenize.s
;;
;; This set of procedures reads a data file and converts the
;; contents of the file to a tokenized list. These procedures are
;; specific for transforming the input from the data file to a
;; tokenized list of symbols representing a system of Boolean
;; equations. The user is prompted for a filename.
;; GET-TOKENIZED-LIST returns a tokenized list of symbols.
;;
;;;;;;;;;;;;;;;

;; (GET-TOKENIZED-LIST)
;;
;; -- GET-TOKENIZED-LIST call GET-LIST-OF-CHARACTERS which returns a
;;   list of characters that were read from a file.
;; -- The CHARACTER-LIST is passed to TOKENIZE which returns a
;;   a properly TOKENIZED-LIST.
;; -- REPLACE-JUXTAPOSITIONS replaces juxtapositions of symbols with
;;   '* symbols.

(define (get-tokenized-list)
  (let* ( (character-list (get-list-of-characters))
         (tokenized-list (tokenize character-list))
         (new-tokenized-list (replace-juxtapositions
                             tokenized-list)) )
    new-tokenized-list))

;; (GET-LIST-OF-CHARACTERS)
;;
;; -- GET-LIST-OF-CHARACTERS gets a string of characters from a file,
;;   converts the string to a list, and returns the list.
;; -- GET-FILENAME prompts the user for the filename. The
;;   INPUT-FILENAME is passed to CALL-WITH-INPUT-FILE along with
;;   GET-STRING-FROM-FILE.
;; -- CALL-WITH-INPUT-FILE returns a string of characters to
;;   INPUT-STRING.
;; -- STRING->LIST converts the string to a list of characters.

(define (get-list-of-characters)
  (let* ( (input-filename (get-filename))
         (input-string (call-with-input-file input-filename
                                           get-string-from-file))
         (input-list (string->list input-string)) )
    (newline)
    input-list))

```

```

;; (GET-FILENAME)
;;
;; -- GET-FILENAME prompts the user for a filename, accepts the
;;     filename from the keyboard, and returns it.

(define (get-filename)
  (newline)
  (newline)
  (writeln "**** Enter the filename of your input file ****")
  (newline)
  (newline)

  (display "Enter the input filename --> ")
  (read-line) )



;; (GET-STRING-FROM-FILE p)
;;
;; Parameters:
;;   p - the port for the file to be read
;;
;; -- GET-STRING-FROM-FILE reads a file and returns a string which
;; includes all of the characters in the file.
;; -- READ-LINE is used to read one line of the file at a time
;; returning a string to INPUT-STRING.
;; -- After each line is read, a check is made to determine if
;; INPUT-STRING is the eof object. If it is, then a null string is
;; returned. Otherwise, a #\newline is STRING-APPENDED to
;; INPUT-STRING to form STRING-WITH-NEWLINE.
;; -- GET-STRING-FROM-FILE is called recursively to build the string
;; which consists of all characters in the input file.

(define (get-string-from-file p)
  (let ( (input-string (read-line p)) )
    (if (eof-object? input-string)
        ""
        (let* ( (newline-string (make-string 1 #\newline))
                (string-with-newline
                  (string-append input-string newline-string)) )
          (string-append string-with-newline
                        (get-string-from-file p)) ))))




;; (TOKENIZE character-list)
;;
;; Parameter:
;;   character-list - a list of characters contained in the

```

```

;;
      input file
;;
;;
-- TOKENIZE takes a character list and returns a tokenized list
-- in which the only elements are symbols, valid tokens, and
-- newline markers.
-- TOKENIZE scans one character at a time until a letter or digit,
-- token, or a newline is reached. Other characters are skipped.
-- When a letter or digit is reached, GET-SYMBOL returns the
-- NEW-SYMBOL. GET-REST-OF-LIST returns the REST-OF-LIST.
-- TOKENIZE is then called recursively to tokenize the
-- REST-OF-LIST.
-- When a token is reached, the token character is enclosed in a
-- list. Then LIST->STRING and STRING->SYMBOL are called to
-- convert the character to a TOKEN-SYMBOL. TOKENIZE is then
-- called recursively on the CDR of CHARACTER-LIST to tokenize the
-- rest of the list.
-- When a newline is detected, the same process is used as when a
-- token is reached. However, the newline character is replaced
-- with #\. to denote the end of a line.

(define (tokenize character-list)
  (if (null? character-list)
      '()
      (let ( (first-char (car character-list)) )
            ; first-char is a letter or digit
        (cond ( (or (letter? first-char)
                    (digit? first-char))
                (let ( (new-symbol (get-symbol character-list))
                      (rest-of-list (get-rest-of-list
                                      character-list)) )
                  (cons new-symbol (tokenize rest-of-list))) )

            ; first-char is a token
            ( (token? first-char)
              (let ( (token-symbol
                      (string->symbol (list->string
                                      (list first-char)))) )
                  (cons token-symbol
                        (tokenize (cdr character-list)))) )

            ; first-char is a #\newline
            ( (eq? first-char #\newline)
              (let ( (token-symbol
                      (string->symbol (list->string
                                      (list #\.)))) )
                  (cons token-symbol
                        (tokenize (cdr character-list)))) )

            ; all other cases
            ( else

```

```

(tokenize (cdr character-list)) ))))

;; (GET-SYMBOL character-list)
;;
;; Parameter:
;;   character-list - a list of characters, the first
;;                     part of which comprise a valid symbol
;;
;; -- GET-SYMBOL gets the characters that comprise a symbol, and
;; converts the characters to a symbol which is then returned.
;; -- It is assumed that the leading characters of CHARACTER-LIST
;; comprise a valid symbol.
;; -- GET-SYMBOL-LIST is returns the sublist of CHARACTER-LIST that
;; consists of the characters that comprise the symbol.
;; -- CONVERT-TO-UPPER-CASE converts the SYMBOL-LIST to upper case
;; to eliminate language quirks involved when dealing with
;; a mix of upper and lower case.
;; -- LIST->STRING and STRING->SYMBOL converts REVISED-SYMBOL-LIST
;; into a valid symbol which is then returned.

(define (get-symbol character-list)
  (let* ( (symbol-list (get-symbol-list character-list))
          (revised-symbol-list (convert-to-upper-case symbol-list)) )

    (string->symbol (list->string revised-symbol-list)))))

;; (GET-SYMBOL-LIST character-list)
;;
;; Parameters:
;;   character-list - a list of characters, the first
;;                     part of which comprise a valid symbol
;;
;; -- GET-SYMBOL-LIST returns a list consisting of the characters that
;; comprise a symbol.
;; -- It is assumed that the beginning characters of the initial input
;; list have been predetermined to be a symbol.
;; -- GET-SYMBOL-LIST scans the CHARACTER-LIST by calling itself
;; recursively until either a token character, space, or newline
;; character is reached.

(define (get-symbol-list character-list)
  (let ( (first-char (car character-list)) )
    (if (or (token? first-char)
            (eq? #\space first-char)
            (eq? #\newline first-char))
        '()
        (cons first-char (get-symbol-list (cdr character-list))))))


```

```

;; (CONVERT-TO-UPPER-CASE character-list)
;;
;; Parameter:
;;   character-list - a list of characters
;;
;; -- CONVERT-TO-UPPER-CASE takes a list of characters and returns a
;;   list in which all lower case characters are converted to upper
;;   case characters.
;; -- CHAR-LOWER-CASE? is used to determine if a character needs to be
;;   converted.  CHAR-UPCASE converts the lower case character to
;;   upper case.

(define (convert-to-upper-case character-list)
  (if (null? character-list)
      '()
      (if (char-lower-case? (car character-list))
          (cons (char-upcase (car character-list))
                (convert-to-upper-case (cdr character-list)))
          (cons (car character-list)
                (convert-to-upper-case (cdr character-list))))))

;;
;; (CHAR-LOWER-CASE? input-char)
;;
;; Parameter:
;;   input-char - a single character
;;
;; -- CHAR-LOWER-CASE? is a predicate procedure for determining
;;   whether a character is lower case.

(define (char-lower-case? input-char)
  (and (char>=? input-char #\a)
       (char<=? input-char #\z)) )

;;
;; (GET-REST-OF-LIST character-list)
;;
;; Parameter:
;;   character-list - a list of characters
;;
;; -- GET-REST-OF-LIST accepts a list and returns a sublist removing
;;   the characters comprising a symbol at the head of the
;;   CHARACTER-LIST.
;; -- TOKEN? is used to determine if a character is a token.

(define (get-rest-of-list character-list)

```

```

(let ( (first-char (car character-list)) )
  (cond ( (token? first-char)
            character-list )
        ( (eq? #\space first-char)
            character-list )
        ( (eq? #\newline first-char)
            character-list )
        ( else
            (get-rest-of-list (cdr character-list)) ))))

;; (LETTER? input-char)
;;
;; Parameter:
;;   input-char - a character
;;
;; -- LETTER? is a predicate procedure for determining whether a
;;   character is a letter.

(define (letter? input-char)
  (or (and (char>=? input-char #\a)
            (char<=? input-char #\z))
      (and (char>=? input-char #\A)
            (char<=? input-char #\Z)))) )

;; (DIGIT? input-char)
;;
;; Parameter:
;;   input-char - a character
;;
;; -- DIGIT? is a predicate procedure for determining whether a
;;   character is a digit.

(define (digit? input-char)
  (and (char>=? input-char #\0)
       (char<=? input-char #\9))) )

;; (TOKEN? input-char)
;;
;; Parameter:
;;   input-char - a character
;;
;; -- TOKEN? is a predicate procedure for determining whether a
;;   character is a token.

(define (token? input-char)

```

```

(or (char=? input-char #\+)
    (char=? input-char #\*)
    (char=? input-char #\()
    (char=? input-char #\))
    (char=? input-char #\=)
    (char=? input-char #\')
    (char=? input-char #\`)
    (char=? input-char #\<)
    (char=? input-char #\!)) )

;; (REPLACE-JUXTAPOSITIONS lst)
;;
;; Parameter:
;;   lst - a list of tokenized symbols
;;
;; -- REPLACE JUXTAPOSITIONS replaces juxtapositions of symbols by
;;   inserting the '*' character between an appropriate juxtaposition
;;   of symbols.
;; -- Juxtapositions replaced are as follows:
;;
;;       symbol1 symbol2 --> symbol1 * symbol2
;;       symbol |(|     --> symbol * |(| 
;;       |)| symbol      --> |)| * symbol
;;       |)| |(|        --> |)| * |(| 
;;       |'| |(|        --> |'| * |(| 
;;       |'| symbol      --> |'| * symbol
;;       symbol `         --> symbol * ` 
;;       |)| `           --> |)| * `

(define (replace-juxtapositions lst)
  (cond ( (null? lst)
          '())
        ( (null? (cdr lst))
          lst )
        ( else
          (let ( (first-symbol (car lst))
                 (second-symbol (cadr lst)) )

            (if (or (and (good-symbol? first-symbol)
                         (good-symbol? second-symbol))
                    (and (good-symbol? first-symbol)
                         (eq? '|(| second-symbol)))
                    (and (eq? '|)| first-symbol)
                         (good-symbol? second-symbol))
                    (and (eq? '|)| first-symbol)
                         (eq? '|(| second-symbol)))
                    (and (eq? '|'| first-symbol)
                         (eq? '|(| second-symbol))))
```

```

        (and (eq? '|' first-symbol)
              (good-symbol? second-symbol))
        (and (good-symbol? first-symbol)
              (eq? '-' second-symbol))
        (and (eq? '|) first-symbol)
              (eq? '-' second-symbol)))
      (cons first-symbol
            (cons '*
                  (replace-juxtapositions (cdr lst))))
      (cons first-symbol
            (replace-juxtapositions (cdr lst))))) )))

;; (GOOD-SYMBOL? input-symbol)
;;
;; Parameter:
;;   input-symbol - a symbol
;;
;; -- GOOD-SYMBOL? is a predicate for determining that a symbol is not
;;   a token, i.e., a good symbol.
;; -- GOOD-SYMBOL? is used for clarity in REPLACE-JUXTAPOSITIONS.

(define (good-symbol? input-symbol)
  (not (token-symbol? input-symbol)))

;; (TOKEN-SYMBOL? input-symbol)
;;
;; Parameter:
;;   input-symbol - a symbol
;;
;; -- TOKEN-SYMBOL? is a predicate procedure for determining whether a
;;   symbol is a valid token.

(define (token-symbol? input-symbol)
  (or (eq? input-symbol '|(| )
        (eq? input-symbol '|)| )
        (eq? input-symbol '|'| )
        (eq? input-symbol '|.| )
        (eq? input-symbol '* )
        (eq? input-symbol '+ )
        (eq? input-symbol '! )
        (eq? input-symbol '- )
        (eq? input-symbol '= )
        (eq? input-symbol '< )
        (eq? input-symbol 'AND)
        (eq? input-symbol 'OR)
        (eq? input-symbol 'XOR)
        (eq? input-symbol 'NOT)

```

```
(eq? input-symbol 'EQ)
(eq? input-symbol 'LE)
(eq? input-symbol 'GE)) )
```

```

;;;;;;
;;          Filename: PREFIXER.S
;;
;; These procedures are used to transform a tokenized input list
;; into a list which is in prefix form. This list may either be a
;; system of equations, or it may be a Boolean formula. The
;; tokenized input list is of the form:
;;
;; ( D = B || .|| E = ||( A * D ||) || .|| F = ||( C * D ||)
;;   || .|| Z = ||( E * F ||) || .|| )
;;
;; The list returned by TOKEN->PREFIX is:
;;
;; ( (EQ D (NOT B)) (EQ E (NOT (* A D))) (EQ F (NOT (* C D)))
;;   (EQ Z (NOT (* E F)))) )
;;;;;;
;;
;; (TOKEN->PREFIX lst)
;;
;; Parameters:
;;   lst - accepts an input list of tokens forming a Boolean equation
;;
;; -- TOKEN->PREFIX accepts an input list of tokens and puts the
;;   list into prefix form.
;; -- GET-LINE-FROM-LIST takes a line from the input list of tokens,
;;   and processes this line. For example, if the line is an
;;   equality such as x = a + b, the line is converted to the
;;   form: (EQ X (+ A B)). If the input line is a formula, then the
;;   line is transformed into prefix form. The end of a line is
;;   demarcated by the newline (||) token. Thus, GET-LINE-FROM-LIST
;;   assumes that all tokens prior to the first occurrence of || are
;;   part of the line to be processed. GET-RIGHT-PART returns all
;;   tokens after the first occurrence of || symbol.
;;   TOKEN->PREFIX then calls itself recursively to process
;;   REST-OF-LIST. The recursive calls end when REST-OF-LIST is nil.
;; -- The LINE processed by GET-LINE-FROM-LIST is appended to the
;;   PREFIXED-LIST returned by the recursive call to TOKEN->PREFIX.
;;   Thus, TOKEN->PREFIX returns a list of LINES produced by the
;;   successive calls to GET-LINE-FROM-LIST via TOKEN->PREFIX, i.e.,
;;   lines are appended together to form a single prefix expression.

(define (token->prefix lst)
  (if (null? lst)
      '()
      ; get a line and process it, then get the rest-of-list
      (let ( (line (get-line-from-list lst))
            (rest-of-list (get-right-part lst '||)) )
        (if (null? line)
            '()
            ; make recursive call to get prefixed-list and append it
            (let ( (prefixed-list (token->prefix rest-of-list)) )

```

```

(if (not (null? prefixed-list))
    (append (list line) prefixed-list)
    (list line) )))))

;; (GET-LINE-FROM-LIST lst)
;;
;; Parameters:
;;   lst - accepts an input list of tokens forming a Boolean equation
;;
;; -- GET-LINE-FROM-LIST takes a sublist from the tokenized symbol
;;   list up to but not including the first newline marker (|.|.
;;
;; -- The procedure assumes that each line has either an equality
;;   or implication operator, or is a Boolean formula.
;;
;; -- If the line has an equality or implication operator, the line is
;;   decomposed appropriately into a left and right side, put into
;;   prefix form, and reassembled appropriately.
;;
;;   GET-LEFT-PART-OF-LINE and GET-RIGHT-PART-OF-LINE decompose LINE
;;   and put the respective parts into prefix form. ASSEMBLE-LINE
;;   puts the parts back together into prefix form.
;;
;; -- If the line is a Boolean formula, signified by the lack of an
;;   equality or implication operator, then the formula is simply put
;;   into prefix form by MAKE-PREFIX-FORM.

(define (get-line-from-list lst)
  (let ( (line (get-left-part lst '|.|)) )
        ; the line contains an equality
    (cond ( (member '= line)
            (let ( (left-part (get-left-part-of-line line '=))
                  (right-part (get-right-part-of-line line '=')) )
              (assemble-line 'EQ left-part right-part)) )
          ; the line contains an implication
        ( (member '< line)
            (let* ( (left-part (get-left-part-of-line line '<))
                  (right-part (get-right-part-of-line line '<)) )
              (assemble-line 'LE left-part right-part)) )
          ; the line is a Boolean formula
        ( else
          (make-prefix-form line) ))))

;; (GET-LEFT-PART-OF-LINE lst input-symbol)
;;
;; Parameters:
;;   lst - a list of tokens which includes input-symbol
;;   input-symbol - the symbol, to the left of which, all symbols
;;                 will be returned in prefix form
;;
;; -- GET-LEFT-PART-OF-LINE returns the sublist of the input list

```

```
; ; which includes all elements up to, but not including the first
; ; occurrence of the input symbol. The returned list is in prefix
; ; form.
; ; -- GET-LEFT-PART returns the sublist of the input list which
; ; includes all elements up to, but not including the first
; ; occurrence of input symbol.
; ; -- The returned list is put into prefix form by MAKE-PREFIX-FORM.
```

```
(define (get-left-part-of-line lst input-symbol)
  (make-prefix-form (get-left-part lst input-symbol)))
```

```
; ; (GET-RIGHT-PART-OF-LINE lst input-symbol)
; ;
; ; Parameters:
; ;   lst - a list of tokens which includes input-symbol
; ;   input-symbol - the symbol, to the right of which, all symbols
; ;                 will be returned in prefix form
; ;
; ; -- GET-RIGHT-PART-OF-LINE returns the sublist of the input list
; ; which includes all elements after, but not including the first
; ; occurrence of the input symbol. The returned list is in prefix
; ; form.
; ; -- GET-RIGHT-PART returns the sublist of the input list which
; ; includes all elements after, but not including the first
; ; occurrence of input symbol.
; ; -- The returned list is put into prefix form by MAKE-PREFIX-FORM.
```

```
(define (get-right-part-of-line lst input-symbol)
  (make-prefix-form (get-right-part lst input-symbol)))
```

```
; ; (ASSEMBLE-LINE token-symbol left-part right-part)
; ;
; ; Parameters:
; ;   token-symbol - the symbol to be placed at the front of the list
; ;   left-part - a list which is either a single element or a list
; ;               in prefix form
; ;   right-part - a list which is either a single element or a list
; ;               in prefix form
; ;
; ; -- ASSEMBLE-LINE assembles prefixed expressions such that the
; ; first element is the token-symbol, and the left- and right-parts
; ; are either prefixed or single element lists.
; ; -- The components are assembled differently depending on the type
; ; of input lists. The assembled list should not include single
; ; element lists, but rather symbols.
```

```
(define (assemble-line token-symbol left-part right-part)
```

```

; both left- and right-part are lists with a single element
(cond ( (and (single-element-list? left-part)
              (single-element-list? right-part))
          (cons token-symbol (append left-part right-part)) )
      ;left-part is a single elem, right-part is in prefix form
      ( (and (single-element-list? left-part)
              (not (null? right-part)))
          (cons token-symbol (append left-part (list right-part)))) )
      ; left-part is in prefix form, right-part is single element
      ( (single-element-list? right-part)
          (cons token-symbol (append (list left-part) right-part)) )
      ; all other cases
      ( else
          (cons token-symbol (append (list left-part)
                                      (list right-part)))) )))

;; (MAKE-PREFIX-FORM lst)
;;
;; Parameters:
;;   lst - an input list of tokens
;;
;;
;; -- MAKE-PREFIX-FORM accepts an input list and puts it into prefix
;; form.
;; -- Parenthetical expressions are broken into sublists by
;; GET-INNER-PART and are put into prefix form by a recursive call
;; to MAKE-PREFIX-FORM. Then the portion of the input list which
;; occurs prior to and after the parenthetical expression, as well
;; as the prefix form of the expression, is reassembled by
;; REASSEMBLE-LIST to form the COMPLETE-LIST. COMPLETE-LIST is
;; used as the parameter to make a recursive call to
;; MAKE-PREFIX-FORM.
;; -- Both prefix and postfix complementation are handled by
;; MAKE-PREFIX-FORM. The symbol to be complemented is replaced by
;; the appropriate sublist which is put into prefix form.
;; GET-LEFT-PART and GET-LEFT-COMP-PART get the part of the list
;; prior to the symbol to be complemented for the prefix and
;; postfix complementation cases, respectively. The LEFT-PART, the
;; input list, and the complementation symbol are then passed to
;; COMP->PREFIX which puts the expression to be complemented into
;; prefix form, reassembles the list, and recursively calls
;; MAKE-PREFIX-FORM.
;; -- Expressions with infix operators (+, *, and !) put into prefix
;; form by INFIX->PREFIX. INFIX->PREFIX disassembles a list into
;; left and right sublists, each of which is put into prefix form.
;; Then the sublists are assembled with the operator in prefix
;; form. The order in which the operators are put into prefix form
;; is based on precedence, with the least priority operator put
;; into prefix form first, i.e., OR, then AND, and finally XOR.

```

```

(define (make-prefix-form lst)
    ; the case of a parenthetical sub-expression
  (cond ( (member '|(| lst)
    (let* ( (inner-part (get-inner-list lst))
            (inner-in-prefix-form (make-prefix-form
                                      inner-part))
            (left-part (get-left-part lst '|(| ))
            (right-part (get-right-outer-list lst))
            (complete-list
              (reassemble-list left-part
                                inner-in-prefix-form
                                right-part)) )
      (make-prefix-form complete-list)) )
    ; the prefixed complementation case
  ( (member '|-' lst)
    (let ( (left-part (get-left-part lst '|-)) )
      (comp->prefix lst '|-' left-part)) )
    ; the postfix complementation case
  ( (member '|)'| lst)
    (let ( (left-part (get-left-comp-part lst)) )
      (comp->prefix lst '|)'| left-part)) )
    ; the infix OR case
  ( (member '+ lst)
    (infix->prefix lst '+ '+) )
    ; the infix OR case
  ( (member 'OR lst)
    (infix->prefix lst 'OR '+) )
    ; the infix AND case
  ( (member '* lst)
    (infix->prefix lst '* '*) )
    ; the infix AND case
  ( (member 'AND lst)
    (infix->prefix lst 'AND '*) )
    ; the infix XOR case
  ( (member '! lst)
    (infix->prefix lst '! '!)) )
    ; the infix XOR case
  ( (member 'XOR lst)
    (infix->prefix lst 'XOR '!)) )
    ; otherwise, return the list
  ( else lst ) )))

;; (GET-INNER-LIST lst)
;;
;; Parameters:
;;   lst - a list of tokens
;;
;; -- GET-INNER-LIST accepts a list of tokens which includes a

```

```

;;      parenthetical expression and returns the sublist of tokens that
;;      make up the expression.
;; -- The enclosing parentheses are not returned, however, nested
;;      parenthetical expressions are included to include parentheses
;;      symbols.
;; -- GET-RIGHT-PART returns the sublist of the input list which
;;      includes all symbols after the first occurrence of '|('.
;;      DELETE-RIGHT-PART returns the sublist of REST-OF-LIST which
;;      includes all symbols prior to the appropriate closing
;;      parentheses symbol.

(define (get-inner-list lst)
  (let* ( (rest-of-list (get-right-part lst '|(| ))
          (inner-list (delete-right-part rest-of-list 0)) )
         inner-list))

;; (DELETE-RIGHT-PART lst nesting-level)
;;
;; Parameters:
;;   lst - a tokenized sublist of which all to the left of, and
;;         including, a left parentheses has been removed
;;   nesting-level - the level at which parenthetical expressions are
;;                  nested (the top level equals 0)
;;
;; -- DELETE-RIGHT-PART is a helping procedure for GET-INNER-LIST.
;;   It deletes the right part of the list after, but not including,
;;   the enclosing right parentheses.
;; -- The level of nesting of parenthetical expressions is monitored
;;   by tracking the nesting level.

(define (delete-right-part lst nesting-level)
  (if (null? lst)
      '()
      (let ( (first-symbol (car lst))
             (rest-of-list (cdr lst)) )
        ; a nested parenthetical expression is detected,
        ; add 1 to nesting level and make a recursive call
        (cond ( (eq? first-symbol '|(| )
                 (cons first-symbol
                       (delete-right-part (cdr lst)
                                         (1+ nesting-level))) )
               ; the right parentheses is detected, return nil
               ( (and (eq? first-symbol '|)| )
                   (eq? nesting-level 0)))
               '())
              ; a right parentheses of a subexpression is
              ; detected, decrement nesting level by one
              ( (eq? first-symbol '|)| )
              (cons first-symbol

```

```

        (delete-right-part (cdr lst)
                           (-1+ nesting-level))) )
; save the current symbol and call recursively
( else
  (cons first-symbol
        (delete-right-part (cdr lst)
                           nesting-level) )))))

```

;; (GET-RIGHT-OUTER-LIST lst)

;; Parameters:

;; lst - a tokenized list

;; -- GET-RIGHT-OUTER-LIST returns the sublist consisting of all elements after a parenthetical expression. When this procedure is called, the initial left parentheses has been detected, and all elements to the right of the respective right parentheses must be returned.

;; -- GET-RIGHT-PART returns the sublist after the left parentheses.

;; -- GET-RIGHT-OUTER-LIST-1 is called to extract the sublist from REST-OF-LIST which is all tokens after the right enclosing parentheses.

```

(define (get-right-outer-list lst)
  (let* ( (rest-of-list (get-right-part lst '|()))
          (right-part (get-right-outer-list-1 rest-of-list 0)) )
    right-part))

```

;; (GET-RIGHT-OUTER-LIST-1 lst nesting-level)

;; Parameters:

;; lst - a tokenized sublist of which all to the left of, and including, a left parentheses has been removed

;; nesting-level - the level at which parenthetical expressions are nested (the top level equals 0)

;; -- GET-RIGHT-OUTER-LIST-1 is a helping procedure for GET-RIGHT-OUTER-LIST.

;; -- GET-RIGHT-OUTER-LIST-1 scans the input list until a right parentheses is detected and the nesting level is 0. Then, the sublist to the right of the parentheses is returned.

;; -- The level of nesting of subexpressions is monitored by nesting-level.

```

(define (get-right-outer-list-1 lst nesting-level)
  (let* ( (first-char (car lst))
          (rest-of-list (cdr lst)) )

```

```

; the right enclosing parentheses is detected
(cond ( (and (eq? first-char '|)|
              (eq? nesting-level 0))
        (cdr lst)
        ; a nested parenthetical subexpression is detected, add
        ; one to the nesting level and make a recursive call
        ( (eq? first-char '|(|)
            (get-right-outer-list-1 (cdr lst) (1+ nesting-level)) )
        ; a right parentheses is detected for a subexpression
        ; decrement nesting level by 1 and make a recursive call
        ( (eq? first-char '|)|)
            (get-right-outer-list-1 (cdr lst) (-1+ nesting-level)) )
        ; otherwise, make a recursive call
        ( else
            (get-right-outer-list-1 (cdr lst) nesting-level) ))))

;; (COMP->PREFIX lst input-symbol left-part)
;;
;; Parameters:
;;   lst - a tokenized list in which a complement symbol was detected
;;   input-symbol - the complement symbol detected (prefix or
;;                 postfix)
;;   left-part - the sublist of lst which includes all symbols prior
;;               to the symbol to be complemented (postfix) or the
;;               prior to the complement symbol (prefix)
;;
;; -- COMP->PREFIX is a helping procedure for MAKE-PREFIX-FORM.
;; -- COMP->PREFIX passes the input list and the input-symbol to
;;   GET-COMPLEMENTED-PART which returns the complemented expression.
;;   MAKE-COMP-PREFIX-FORM returns the prefix form of
;;   COMPLEMENTED-PART.
;; -- GET-ALL-AFTER-COMP-PART returns the sublist which is all tokens
;;   after the complemented expression.
;; -- REASSEMBLE-LIST takes the LEFT-PART, the complemented symbol in
;;   prefix form, and the RIGHT-PART and forms the COMPLETE-LIST.
;;   MAKE-PREFIX-FORM is then called recursively to put COMPLETE-LIST
;;   into prefix form.

(define (comp->prefix lst input-symbol left-part)
  (let* ( (complemented-part (get-complemented-part lst
                                                    input-symbol))
          (comp-part-in-prefix-form
            (make-comp-prefix-form complemented-part))
          (right-part (get-all-after-comp-part lst input-symbol))
          (complete-list (reassemble-list left-part
                                         comp-part-in-prefix-form
                                         right-part)) )
    (make-prefix-form complete-list)))

```

```

;; (GET-LEFT-COMP-PART lst)
;;
;; Parameters:
;;   lst - a tokenized list in which a postfix complement is detected
;;
;; -- GET-LEFT-COMP-PART returns the sublist containing all elements
;;   to the left of the symbol to be complemented. This procedure is
;;   used only in the postfix complementation case.
;; -- The procedure calls itself recursively until the postfix
;;   complement is detected.

(define (get-left-comp-part lst)
  (let ( (first-char (car lst)) )
    (if (eq? '|`| (cadr lst))
        '()
        (cons first-char (get-left-comp-part (cdr lst))))))

;;
;; (GET-COMPLEMENTED-PART lst input-symbol)
;;
;; Parameters:
;;   lst - a token list in which a complement token has been detected
;;   input-symbol - the token symbol that was detected
;;
;; -- GET-COMPLEMENTED-PART returns the symbol that is to be
;;   complemented, handling both postfix and prefix complemented
;;   cases.

(define (get-complemented-part lst input-symbol)
  (if (eq? '` input-symbol)
      ; prefix case
      (if (eq? '` (car lst))
          (cadr lst)
          (get-complemented-part (cdr lst) input-symbol))
      ; postfix case
      (let ( (first-char (car lst)) )
        (if (eq? '|`| (cadr lst))
            first-char
            (get-complemented-part (cdr lst) input-symbol)))))

;;
;; (MAKE-COMP-PREFIX-FORM input-symbol)
;;
;; Parameters:
;;   input-symbol - the symbol to be complemented (in prefix form)
;;
;; -- MAKE-COMP-PREFIX-FORM returns a list with the input symbol in a

```

```

;;      complemented, prefix form.

(define (make-comp-prefix-form input-symbol)
  '(NOT ,input-symbol))

;; (GET-ALL-AFTER-COMP-PART lst input-symbol)
;;
;; Parameters:
;;   lst - a token list in which a complement token has been detected
;;   input-symbol - the token symbol that was detected
;;
;; -- GET-ALL-AFTER-COMP-PART returns the part of the list after the
;;   symbol to be complemented.
;; -- Both prefix complement case `a, and the postfix case a' are
;;   handled.
;; -- Neither the symbol, nor the complement symbol is returned with
;;   the sublist.

(define (get-all-after-comp-part lst input-symbol)
  (if (eq? '-' input-symbol)
      ; prefix case
      (if (eq? '-' (car lst))
          (cddr lst)
          (get-all-after-comp-part (cdr lst) input-symbol))
      ; postfix case
      (if (eq? '|'| (cadr lst))
          (cddr lst)
          (get-all-after-comp-part (cdr lst) input-symbol)))))

;; (REASSEMBLE-LIST left-part inner-part right-part)
;;
;; Parameters:
;;   left-part - a list
;;   inner-part - a list
;;   right-part - a list
;;
;; -- REASSEMBLE-LIST reassembles a list consisting of left, inner,
;;   and right sublists. It is used after the symbol list is
;;   disassembled when making into prefix form either parenthetical
;;   or complemented expressions.

(define (reassemble-list left-part inner-part right-part)
  (append left-part (append (list inner-part) right-part)))

;; (SINGLE-ELEMENT-LIST? obj)

```

```

;;
;; Parameter:
;;   obj - an arbitrary object
;;
;; -- SINGLE-ELEMENT-LIST? is a predicate procedure for determining
;;   whether a list is a list that consists of a single element.
;; -- If so, #T is returned. If not, nil is returned.

(define (single-element-list? obj)
  (cond ( (not (list? obj))
           '() )
        ( (null? obj)
           '() )
        ( else
           (eq? '() (cdr obj)) )))

;;
;; (LIST? obj)
;;
;; Parameter:
;;   obj - an arbitrary object
;;
;; -- LIST? is a predicate procedure for determining whether an object
;;   is a list. If so, #T is returned. Else, nil is returned.

(define (list? obj)
  (or (pair? obj)
      (null? obj)))

;;
;; (INFIX->PREFIX lst input-symbol output-symbol)
;;
;; Parameters:
;;   lst - a tokenized list which contains an infix operator
;;   input-symbol - the infix operator in the list
;;   output-symbol - the prefix operator that will replace the
;;                 input-symbol (if different from input-symbol)
;;
;; -- INFIX->PREFIX is a helping procedure for MAKE-PREFIX-FORM.
;; -- In the procedure, the sublist prior to the operator is obtained
;;   by GET-LEFT-PART. This sublist is then made into prefix form by
;;   MAKE-PREFIX-FORM.
;; -- Likewise, the sublist after the operator is obtained by
;;   GET-RIGHT-PART and put into prefix form by MAKE-PREFIX-FORM.
;; -- ASSEMBLE-LINE is called to return the prefix form of the infix
;;   expression in which the output-symbol, the operator, the
;;   LEFT-IN-PREFIX-FORM, and the RIGHT-IN-PREFIX-FORM are put
;;   together to form a COMPLETE-LIST. COMPLETE-LIST is then
;;   returned.

```

```

(define (infix->prefix lst input-symbol output-symbol)
  (let* ( (left-part (get-left-part lst input-symbol))
          (left-in-prefix-form (make-prefix-form left-part))
          (right-part (get-right-part lst input-symbol))
          (right-in-prefix-form (make-prefix-form right-part))
          (complete-list (assemble-line output-symbol
                                         left-in-prefix-form
                                         right-in-prefix-form)) )
    complete-list))

;; (GET-LEFT-PART lst symbol)
;;
;; Parameters:
;;   lst      - a an arbitrary list of symbols
;;   symbol  - the symbol, to the left of which, all elements
;;             are returned
;;
;; -- GET-LEFT-PART returns the sublist of the input list which
;; includes all elements up to, but not including the first
;; occurrence of the symbol.
;; -- It is assumed that the symbol is an element of the list,
;; -- If the input list is null, '() is returned.
;; -- GET-LEFT-PART calls itself recursively until symbol is
;; found in the list. The returned list is built from all symbols
;; which precede symbol.

(define (get-left-part lst symbol)
  (cond ( (null? lst)
           '() )
        ( (eq? (car lst) symbol)
           '() )
        ( else
           (cons (car lst) (get-left-part (cdr lst) symbol)) )))

;;
;; (GET-RIGHT-PART lst symbol)
;;
;; Parameters:
;;   lst      - a an arbitrary list of symbols
;;   symbol  - the symbol, to the left of which, all elements
;;             are returned
;;
;; -- GET-RIGHT-PART returns the sublist of the input list which
;; includes all elements after, but not including the first
;; occurrence of symbol.
;; -- It is assumed that symbol is an element of the list,
;; -- GET-RIGHT-PART calls itself recursively until symbol is

```

```
; found in the list. The returned list is then the sublist
;; which follows symbol.

(define (get-right-part lst input-symbol)
  (if (eq? (car lst) input-symbol)
      (cdr lst)
      (get-right-part (cdr lst) input-symbol)))
```

```

::::::::::::::::::: Filename: tok-vhdl.s ::::::::::::
::: This set of procedures reads a data file which is a dataflow :::
::: view of a VHDL combinational circuit architecture and converts :::
::: the contents of the file to a tokenized list (GET-VHDL-TOKENS). :::
::: If the following VHDL architecture were input to the system: :::
:::
-- architecture declaration for Circuit_Z
architecture DataFlow of Circuit_Z is
    signal E, F : bit;
begin
    F <= not B after 5ns;
    E <= not (A and B);
    Z <= not (E and F);
end DataFlow;
:::
::: The following list would be returned by GET-VHDL-TOKENS:
:::
(F <= NOT B |;| E <= NOT |(| A AND B |)| |;|
Z <= NOT |(| E AND F |)| |;|)
:::
::: Then, VHDL->PREFIX converts this list to the following
::: prefix-form:
:::
((EQ F (NOT B)) (EQ E (NOT (* A B))) (EQ Z (NOT (* E F))))
:::
::: Requires the files: tokenize.fsl, prefixer.fsl
:::::::::::;

;; (GET-VHDL-TOKENS)
;;
;; -- GET-VHDL-TOKENS call GET-LIST-OF-CHARACTERS from file
;;   tokenize.fsl which returns a list of characters that were read
;;   from a file.
;; -- The CHARACTER-LIST is passed to TOKENIZE-VHDL which returns a
;;   a properly TOKENIZED-LIST of all the characters in the input
;;   VHDL description.
;; -- REMOVE-COMMENTS removes all comments from the TOKENIZED-LIST
;;   yielding TOKENIZED-LIST-1.
;; -- After comments are removed, the remaining tokens are valid
;;   identifiers, syntactic symbols, and newlines. REMOVE-NEWLINES
;;   removes newlines from the TOKENIZED-LIST-1 yielding
;;   TOKENIZED-LIST-2.
;; -- The portion of the VHDL description up to and including the
;;   BEGIN statement are irrelevant for the purposes of the
;;   diagnostic system. The same holds for all symbols after and
;;   including the END statement. REMOVE-ARCH-SHELL returns
;;   TOKENIZED-LIST-3 which is all tokens between the BEGIN and END
;;   statements.

```

```

;; -- Finally, REMOVE-AFTER-STATMENTS extracts the "after statements" from
;;   the signal assignment statements. These are irrelevant for the
;;   diagnostic system. TOKENIZED-LIST-4 is then returned.
;; -- An example is given in the file header.

(define (get-vhdl-tokens)
  (let* ( (character-list    (get-list-of-characters))
          (tokenized-list    (tokenize-vhdl character-list))
          (tokenized-list-1  (remove-comments tokenized-list))
          (tokenized-list-2  (remove-newlines tokenized-list-1))
          (tokenized-list-3  (remove-arch-shell tokenized-list-2))
          (tokenized-list-4  (remove-after-stmts tokenized-list-3)) )
    tokenized-list-4))

;; (TOKENIZE-VHDL character-list)
;;
;; Parameters:
;;   character-list - a list of characters
;;
;; -- TOKENIZE-VHDL takes a character list and returns a tokenized
;;   list in which the only elements are VHDL identifier symbols,
;;   compound delimiters, valid single character tokens, and newline
;;   markers.
;; -- IDENTIFIER-CHAR? is called to determine whether a FIRST-CHAR
;;   represents a character that is part of a VHDL identifier.
;;   Appropriate characters include letters, digits, and underline
;;   characters.
;; -- If an identifier character is detected, then GET-VHDL-ID-SYMBOL
;;   gets the rest of the characters corresponding to the identifier,
;;   makes a symbol, and returns it to NEW-ID-SYMBOL.
;;   GET-REST-OF-VHDL-ID-LST returns the REST-OF-LIST which is the
;;   remaining part of the character list with the identifier
;;   characters removed. TOKENIZE-VHDL is then called recursively to
;;   tokenize the REST-OF-LIST.
;; -- VHDL-COMPOUND-DELIMITER? is called to determine if FIRST-CHAR
;;   represents the first character of a VHDL compound delimiter.
;;   These are two-character symbols such as <= or --.
;; -- If a VHDL compound delimiter is detected, then
;;   GET-VHDL-TOK-SYMBOL gets the rest of the characters
;;   corresponding to the symbol, makes the symbol, and returns it to
;;   NEW-TOK-SYMBOL. GET-REST-OF-VHDL-TOK-LST returns the
;;   REST-OF-LIST which is the remaining part of the character list
;;   with the delimiter characters removed. TOKENIZE-VHDL is then
;;   called recursively to tokenize the REST-OF-LIST.
;; -- VHDL-TOKEN-CHAR? is called to determine if FIRST-CHAR
;;   represents a single character VHDL token symbol.
;; -- If a single character VHDL symbol is detected, then
;;   MAKE-VHDL-S-TOK-SYM gets the character corresponding to the
;;   symbol, makes the symbol, and returns it to NEW-TOK-SYMBOL. The

```

```

;;      REST-OF-LIST is nothing more than the remaining part of the
;;      character list with the single character removed. TOKENIZE-VHDL
;;      is then called recursively to tokenize the REST-OF-LIST.
;; -- Newlines are included in the returned list. Spaces are ignored.

(define (tokenize-vhdl character-list)
  (if (null? character-list)
      '()
      (let ((first-char (car character-list)))
        (cond ((identifier-char? first-char)
                (let ((new-id-symbol
                      (get-vhdl-id-symbol character-list))
                     (rest-of-list
                      (get-rest-of-vhdl-id-lst character-list)))
                  (cons new-id-symbol
                        (tokenize-vhdl rest-of-list))))
              ((vhdl-compound-delimiter? first-char
                                         (cadr character-list))
               (let ((new-tok-symbol
                     (get-vhdl-tok-symbol character-list))
                     (rest-of-list
                      (get-rest-of-vhdl-tok-lst character-list)))
                  (cons new-tok-symbol
                        (tokenize-vhdl rest-of-list)) ) )
              ((vhdl-token-char? first-char)
               (let ((new-tok-symbol
                     (make-vhdl-s-tok-sym first-char))
                     (rest-of-list (cdr character-list)) )
                  (cons new-tok-symbol
                        (tokenize-vhdl rest-of-list)) ) )
              ((char=? #\newline first-char)
               (cons first-char
                     (tokenize-vhdl (cdr character-list)))) )
              (else
               (tokenize-vhdl (cdr character-list)) )))))
  ;; (GET-VHDL-ID-SYMBOL character-list)
  ;;
  ;; Parameter:
  ;;   character-list - a list of characters
  ;;
  ;; -- GET-VHDL-ID-SYMBOL calls GET-VHDL-ID-LIST which returns the list
  ;;   of characters comprise an identifier symbol.
  ;;   CONVERT-TO-UPPER-CASE is then called to convert the characters
  ;;   to upper case.

```

```

;; -- The REVISED-SYMBOL-LIST is then converted to a symbol.

(define (get-vhdl-id-symbol character-list)
  (let* ( (symbol-list (get-vhdl-id-list character-list))
          (revised-symbol-list (convert-to-upper-case symbol-list)) )
    (string->symbol (list->string revised-symbol-list)) )

;; (GET-VHDL-ID-LIST character-list)
;;
;; Parameters:
;;   character-list - a list of characters
;;
;; -- GET-VHDL-ID-LIST returns a list consisting of the characters at
;;   the head of the initial CHARACTER-LIST that comprise a VHDL
;;   identifier symbol. The FIRST-CHAR of the initial CHARACTER-LIST
;;   has been predetermined to be a symbol prior to the call to
;;   GET-VHDL-ID-LIST.
;; -- The input CHARACTER-LIST is scanned until either a character
;;   which is not an identifier character or the end of the list has
;;   been reached.

(define (get-vhdl-id-list character-list)
  (let ( (first-char (car character-list))
         (rest-of-list (cdr character-list)) )

    (cond ( (null? rest-of-list)
            (list first-char) )
          ( (identifier-char? first-char)
            (cons first-char (get-vhdl-id-list rest-of-list)) )
          ( else
            '() ))))

;; (GET-REST-OF-VHDL-ID-LST character-list)
;;
;; Parameters:
;;   character-list - a list of characters
;;
;; -- GET-REST-OF-VHDL-ID-LST accepts a list and returns the sublist
;;   removing the characters comprising a symbol at the head of the
;;   input list.
;; -- When a character that is not an identifier character is reached,
;;   the remaining part of the CHARACTER-LIST is returned.

(define (get-rest-of-vhdl-id-lst character-list)
  (if (null? character-list)
      '()

```

```

(let ((first-char (car character-list)))
  (if (not (identifier-char? first-char))
      character-list
      (get-rest-of-vhdl-id-lst (cdr character-list)))))

;; (IDENTIFIER-CHAR? input-char)
;;
;; Parameter:
;;   input-char - a character
;;
;; -- IDENTIFIER-CHAR? is a predicate procedure for testing whether a
;;   character is a valid character for a VHDL identifier. Letters,
;;   digits, and the underline character are all valid identifier
;;   characters.

(define (identifier-char? input-char)
  (or (letter? input-char)
      (digit? input-char)
      (char=? #\_ input-char)))

;; (VHDL-COMPOUND-DELIMITER? char-1 char-2)
;;
;; Parameters:
;;   char-1 - a character
;;   char-2 - a character
;;
;; -- VHDL-COMPOUND-DELIMITER? is a predicate procedure for
;;   determining whether a combination of two characters represents a
;;   VHDL compound delimiter. CHAR-1 is the first character, CHAR-2
;;   the second character of a valid delimiter. -- and <= are valid
;;   delimiters.

(define (vhdl-compound-delimiter? char-1 char-2)
  (or (and (char=? #\= char-1)
            (char=? #\= char-2))
      (and (char=? #\< char-1)
            (char=? #\= char-2)))))

;; (GET-VHDL-TOK-SYMBOL character-list)
;;
;; Parameters:
;;   character-list - a list of characters
;;
;; -- GET-VHDL-TOK-SYMBOL calls GET-VHDL-TOK-LIST which returns the
;;   list of characters comprise a token symbol.

```

```

;; -- The SYMBOL-LIST is then converted to a symbol.

(define (get-vhdl-tok-symbol character-list)
  (let* ( (symbol-list (get-vhdl-tok-list character-list)) )
    (string->symbol (list->string symbol-list))))


;; (GET-VHDL-TOK-LIST character-list)
;;
;; Parameters:
;;   character-list - a list of characters
;;
;; -- GET-VHDL-TOK-LIST returns a list consisting of the characters at
;; the head of the initial CHARACTER-LIST that comprise a VHDL
;; token symbol. The FIRST-CHAR of the initial CHARACTER-LIST has
;; been predetermined to be a symbol prior to the call to
;; GET-VHDL-TOK-LIST.
;;
;; -- The input CHARACTER-LIST is scanned until either a character
;; which is not a token character or the end of the list has been
;; reached.

(define (get-vhdl-tok-list character-list)
  (let ( (first-char (car character-list))
        (rest-of-list (cdr character-list)) )

    (cond ( (null? rest-of-list)
            (list first-char) )
          ( (vhdl-token-char? first-char)
            (cons first-char (get-vhdl-tok-list rest-of-list)) )
          ( else
            '() ))))

;; (GET-REST-OF-VHDL-TOK-LST character-list)
;;
;; Parameters:
;;   character-list - a list of characters
;;
;; -- GET-REST-OF-VHDL-TOK-LST accepts a list and returns the sublist
;; removing the characters comprising a token symbol at the head of
;; the input list.
;;
;; -- When a character that is not a token character is reached,
;; the remaining part of the CHARACTER-LIST is returned.

(define (get-rest-of-vhdl-tok-lst character-list)
  (if (null? character-list)
      '()
      (let ((first-char (car character-list)))
        (if (not (vhdl-token-char? first-char))

```

```

        character-list
        (get-rest-of-vhdl-tok-lst (cdr character-list)))))

;; (VHDL-TOKEN-CHAR? input-char)
;;
;; Parameters:
;;   input-char - a character
;;
;; -- VHDL-TOKEN-CHAR? is a predicate procedure for testing whether a
;;   character is a valid character for a VHDL token.

(define (vhdl-token-char? input-char)
  (or (char=? #\` input-char)
      (char=? #\`\\( input-char)
      (char=? #\\` input-char)
      (char=? #\\`\\( input-char)
      (char=? #\\`\\, input-char)
      (char=? #\\`\\; input-char)
      (char=? #\\`\\: input-char)
      (char=? #\\`\\< input-char)
      (char=? #\\`\\= input-char)))

;; (MAKE-VHDL-S-TOK-SYM character)
;;
;; Parameter:
;;   character - a character
;;
;; -- MAKE-VHDL-S-TOK-SYM takes a character which is a single
;;   character token symbol and converts it to a symbol.

(define (make-vhdl-s-tok-sym character)
  (string->symbol (list->string (list character)) ) )

;; (REMOVE-COMMENTS list-of-tokens)
;;
;; Parameters
;;   list-of-tokens - a list of token symbols
;;
;; -- REMOVE-COMMENTS accepts a list of token symbols and removes all
;;   of the symbols that are associated with a VHDL comment. These
;;   symbols start with a "--" symbol and end with a newline
;;   character.
;; -- The list of symbols is scanned until a -- symbol is detected.
;;   Then, REMOVE-COMMENT-1 is called to remove the symbols
;;   associated with the following comment. REMOVE-COMMENTS then

```

```

;;      calls itself recursively to remove any remaining comments.

(define (remove-comments list-of-tokens)
  (cond ( (null? list-of-tokens)
           '() )
        ; a comment has been detected
        ( (equal? (car list-of-tokens) '--)
           (remove-comments (remove-comments-1 list-of-tokens)) )
        ( else
           (cons (car list-of-tokens)
                 (remove-comments (cdr list-of-tokens))))))

;; (REMOVE-COMMENTS-1 list-of-tokens)
;;
;; Parameters:
;;   list-of-tokens - a list of token symbols
;;
;; -- REMOVE-COMMENTS-1 removes the token symbols associated with a
;; comment. A comment is detected by REMOVE-COMMENTS; then,
;; REMOVE-COMMENTS-1 deletes all characters until a newline
;; character or the end of the list of symbols is reached.
;; -- The rest of the list, after and including the newline character
;; is returned.

(define (remove-comments-1 list-of-tokens)

  (if (or (equal? (car list-of-tokens) #\newline)
          (equal? list-of-tokens '() ))
      list-of-tokens
      (remove-comments-1 (cdr list-of-tokens)))))

;; (REMOVE-NEWLINES list-of-tokens)
;;
;; Parameters:
;;   list-of-tokens - a list of token symbols
;;
;; -- REMOVE-NEWLINES scans the list of tokens removing any newline
;; characters. All other symbols are left in the list which is
;; returned.

(define (remove-newlines list-of-tokens)
  (cond ( (null? list-of-tokens)
           '() )
        ( (equal? (car list-of-tokens) #\newline)
           (remove-newlines (cdr list-of-tokens)) )
        ( else
           (cons (car list-of-tokens)

```

```

(remove-newlines (cdr list-of-tokens))) )))

;; (REMOVE-ARCH-SHELL list-of-tokens)
;;
;; Parameters:
;;   list-of-tokens - a list of token symbols
;;
;; -- REMOVE-ARCH-SHELL removes all tokens from the LIST-OF-TOKENS up
;;   to and including the 'BEGIN symbol. Then REMOVE-ARCH-SHELL-1 is
;;   called to remove all symbols after and including the 'END
;;   symbol.
;; -- The only symbol relevant to the operation of the diagnostic
;;   system are those between the 'BEGIN and 'END symbols in the
;;   VHDL architecture body.

(define (remove-arch-shell list-of-tokens)
  (if (equal? (car list-of-tokens) 'BEGIN)
      (remove-arch-shell-1 (cdr list-of-tokens))
      (remove-arch-shell (cdr list-of-tokens)))))

;;
;; (REMOVE-ARCH-SHELL-1 list-of-tokens)
;;
;; Parameters:
;;   list-of-tokens - a list of token symbols
;;
;; -- REMOVE-ARCH-SHELL-1 accepts a list of token symbols and returns
;;   all of the symbols in a list which occur prior to the 'END
;;   symbol.
;; -- REMOVE-ARCH-SHELL-1 calls itself recursively until the 'END
;;   symbol is detected.

(define (remove-arch-shell-1 list-of-tokens)
  (if (equal? (car list-of-tokens) 'END)
      '()
      (cons (car list-of-tokens)
            (remove-arch-shell-1 (cdr list-of-tokens))))))

;;
;; (REMOVE-AFTER-STMITS list-of-tokens)
;;
;; Parameters:
;;   list-of-tokens - a list of token symbols
;;
;; -- REMOVE-AFTER-STMITS removes the symbols associated with a VHDL
;;   after statement.
;; -- For example, tokens representing the expression:

```

```

;;
; F <= not B after 5ns;
;;
; would be deleted changing the expression to the form:
;;
; F <= not B;
;;
; -- Timing considerations, although necessary for some VHDL models,
; are irrelevant in the diagnostic system.
; -- The list of symbols is scanned until the first 'AFTER symbol
; is detected. Then, REMOVE-AFTER-STMITS-1 is called to remove the
; symbols associated with the detected "after expression."
; -- REMOVE-AFTER-STMITS then calls itself recursively to
; remove any remaining "after expressions."
;

(define (remove-after-stmts list-of-tokens)
  (cond ( (null? list-of-tokens)
           '() )
        ( (equal? (car list-of-tokens) 'AFTER)
           (remove-after-stmts (remove-after-stmts-1 list-of-tokens)))
        ( else
           (cons (car list-of-tokens)
                 (remove-after-stmts (cdr list-of-tokens))))))

;;
; (REMOVE-AFTER-STMITS-1 list-of-tokens)
;;
; Parameters:
;   list-of-tokens - a list of token symbols
;;
; -- REMOVE-AFTER-STMITS-1 removes the token symbols associated with
; an "after expression." An after expression is detected by
; REMOVE-AFTER-STMITS; then, REMOVE-AFTER-STMITS-1 deletes all
; symbols until a semicolon symbol is reached. A semicolon symbol
; denotes the end of the signal assignment statement.
; -- The rest of the list, after and including the semicolon symbol,
; is returned.
;

(define (remove-after-stmts-1 list-of-tokens)
  (if (equal? (car list-of-tokens) '|;|)
      list-of-tokens
      (remove-after-stmts-1 (cdr list-of-tokens))))


;;
; (VHDL->PREFIX vhdl-token-list)
;;
; Parameters:
;   vhdl-token-list - a list of tokens representing VHDL signal
;                     assignment statements of the form:
;
```

```

;;
;          (F <= NOT B |;| E <= NOT (( A AND B )| |;|
;                                Z <= NOT (( E AND F )| |;|)

;;
;; -- VHDL->PREFIX converts the input list to the following
;;    prefix-form:
;;
;;      ((EQ F (NOT B)) (EQ E (NOT (* A B))) (EQ Z (NOT (* E F)))))

;;
;; -- Each set of tokens up to a semicolon symbol represents a single
;;    gate. GET-LEFT-PART from file prefixer.s returns the list of
;;    symbols up to the first occurrence of a semicolon symbol.
;; -- GET-RIGHT-PART from file prefixer.s returns the list of symbols
;;    after the first occurrence of the semicolon symbol.
;; -- MAKE-PREFIX-LIST converts the symbols representing a single gate
;;    to the prefix-form of the gate.
;; -- VHDL->PREFIX calls itself recursively to convert the
;;    REMAINING-EQNS to prefix-form.

(define (vhdl->prefix vhdl-token-list)
  (if (null? vhdl-token-list)
      '()
      (let* ( (first-gate-eqn (get-left-part vhdl-token-list '|;|))
              (remaining-eqns (get-right-part vhdl-token-list '|;|)))
          (cons (make-prefix-list first-gate-eqn)
                (vhdl->prefix remaining-eqns)))))

;;
;; (MAKE-PREFIX-LIST input-list)
;;
;; Parameters:
;;   input-list - a list of tokens of the form:
;;                 E <= NOT (( A AND B )| |
;;                           which represent a single logic gate.
;;
;; -- MAKE-PREFIX-LIST converts the input token list to prefix-form.
;;   For the given example, the list returned would be of the form:
;;
;;           (EQ E (NOT (* A B)))

;;
;; -- It is assumed that a single symbol exists on the left-hand side
;;    of the given signal assignment statement.
;; -- The '<=' symbol is assumed to map directly into an equivalent
;;    'EQ symbol in the mathematical expression representing the gate.
;; -- MAKE-PREFIX-FORM (file prefixer.s) converts the formula
;;    representing the input to the gate to an equivalent prefix-form.
;; -- ASSEMBLE-LINE from file prefixer.s assembles the respective
;;    components into the prefix-form that represents a gate.

(define (make-prefix-list input-list)
  (assemble-line 'EQ

```

```
(car input-list)
(make-prefix-form (caddr input-list)))

;;
;; (GO)
;;
;; -- GO prompts the user for an input VHDL file which is then read
;;   in, tokenized, and converted to prefix-form.

(define (go)
  (newline)
  (writeln "VHDL file to prefix-form conversion utility.")
  (newline)
  (vhdl->prefix (get-vhdl-tokens)))
```

```

;;;;;;
;;          Filename: eqn-gen.s
;;
;; Requires the following files to be loaded: new-sop.s, eqn-gena.s
;; NOTE: eqn-gena.s is a continuation of eqn-gen.s
;;
;; These procedures are used to convert the intermediate format
;; received from the input module to a single Boolean equation in
;; sum of products form. Inserted into this equation is information
;; pertaining to all of the checkpoints nodes within the circuit.
;; For each specific checkpoint a new equation is added to the
;; original equation to form a new equation. As each checkpoint
;; equation is added, the equation is reduced and simplified. After
;; internal nodes are eliminated from the equation, the sum of
;; products equation is simplified and returned. A summary of this
;; process is as follows:
;;
;; 1. Start with a list in intermediate-form.
;;
;; ((EQ E-- (NOT (* A-- B--)))
;;  (EQ F-- (NOT B--))
;;  (EQ Z-- (NOT (* E-- F--))))
;;
;; 2. Determine the internal nodes.
;;
;; (E-- F--)
;;
;; 3. Get the input nodes that do not fanout.
;;
;; (A--)
;;
;; 4. Make the fanout nodes unique.
;;
;; ((EQ E-- (NOT (* A-- B0-)))
;;  (EQ F-- (NOT B1-))
;;  (EQ Z-- (NOT (* E-- F--))))
;;
;; 5. Reduce the prefix form of the list to sum of products
;; (f=0) form.
;;
;; ((E-- A-- B0-) ((E--) A-- (B0-)) ((E--) (A--)) (F-- B1-)
;;  ((F--) (B1-)) (Z-- E-- F--) ((Z--) E-- (F--)) ((Z--) (E--)))
;;
;; 6. Convert the symbols for the input nodes that do not fanout
;; to distinguish them from those that do.
;;
;; ((E-- AX- B0-) ((E--) AX- (B0-)) ((E--) (AX-)) (F-- B1-)
;;  ((F--) (B1-)) (Z-- E-- F--) ((Z--) E-- (F--)) ((Z--) (E--)))
;;
;; 7. Get a list of the nodes which fanout (unique nodes).
;;

```

```

;;
;;      (B0- B1-)
;;
;;
;; 8. Insert the checkpoint equations for the fanout nodes.
;;     After each equation is added, carry out reduction and
;;     elimination of the unnecessary nodes.
;;
;;
;; (((F--) (B--) (B11))((F--) B10 (B11))(F-- B11) (F-- B-- (B10))
;; ((E--) B00 (B01))((E--) (B--) (B01))((E--) (AX-)) (Z-- E-- F--))
;; ((F--) (Z--))((Z--) (E--))( E-- B-- (B00) AX-)(E-- B01 AX-))
;;
;;
;; 9. Get a list of the symbols which represent the input nodes
;;     that do not fanout.
;;
;;
;;      (AX-)
;;
;;
;; 10. Insert the checkpoint equations for the input nodes which do
;;     not have fanouts. Eliminate the unnecessary nodes.
;;
;;
;; (((E--) (A--) (AX1)) ((E--) AX0 (AX1)) (E-- AX1 B01)
;; (E-- A-- (AX0) B01)(E-- AX1 B-- (B00))(E-- A-- (AX0) B-- (B00))
;; ((Z--) (E--)) ((F--) (Z--)) (Z-- E-- F--) ((E--) (B--) (B01)))
;; ((E--) B00 (B01)) (F-- B-- (B10)) (F-- B11) ((F--) B10 (B11))
;; ((F--) (B--) (B11)))
;;
;;
;; 11. Eliminate the internal nodes.
;;
;;
;; (((Z--) B-- (B10)) ((Z--) B11) (Z-- B10 (B11) (A--) (AX1))
;; (Z-- (B--) (B11) (A--) (AX1)) (Z-- B10 (B11) AX0 (AX1))
;; (Z-- (B--) (B11) AX0 (AX1)) (Z-- (B--) (B11) (B01))
;; (Z-- B10 (B11) (B01) B00) (A-- (AX0) (Z--) B01)
;; (A-- (AX0) (Z--) B-- (B00))(AX1 (Z--) B01)(AX1 (Z--) B-- (B00)))
;;
;;
;; 12. Get the checkpoint variables to be used to generate the
;;     constraint-equation. (Get all symbols that end with a zero.)
;;
;;
;;      (B10 AX0 B00)
;;
;;
;; 13. Generate the constraint equation.
;;
;;
;;      ((B10 B11) (AX0 AX1) (B00 B01))
;;
;;
;; 14. Add the constraint equation to the current equation to get
;;     the final equation. Simplify the equation and return it.
;;
;;
;; ((AX1 (Z--) B-- (B00))(AX1 (Z--) B01)(A-- (AX0) (Z--) B-- (B00))
;; (A-- (AX0) (Z--) B01) (Z-- (B--) (B11) (B01))
;; (Z-- (B--) (B11) (A--) (AX1)) ((Z--) B11) ((Z--) B-- (B10))
;; (B10 B11) (Z-- B10 (A--) (AX1)) (AX0 AX1) (Z-- (B--) (B11) AX0)
;; (AX0 Z-- B10) (B00 B01) (B00 Z-- B10))
;;
;
;
```

```

;; For the given circuit, the returned equation would have the ;;
;; following terms: ;;

;;      A-- AX0'B-- BO0'Z--'
;;      A--'AX1'B--'B11'Z--'
;;      AX1 B-- BO0'Z--'
;;      A-- AX0'BO1 Z--'
;;      B--'BO1'B11'Z--'
;;      A--'AX1'B10 Z--'
;;      AX0 B--'B11'Z--'
;;      AX1 BO1 Z--'
;;      B-- B10'Z--'
;;      AX0 B10 Z--'
;;      BO0 B10 Z--'
;;      B11 Z--'
;;      B10 B11
;;      AX0 AX1
;;      BO0 BO1
;;
;; (GENERATE-EQUATION intermediate-format)

;; Parameters:
;;   intermediate-format - a list of the form:
;;
;;   ((EQ E-- (NOT (* A-- B--)))
;;    (EQ F-- (NOT B--))
;;    (EQ Z-- (NOT (* E-- F--))))
;;
;; -- GENERATE-EQUATION accepts a system of equations in
;; INTERMEDIATE-FORMAT and converts them to a single Boolean
;; equation in sum of products form. Inserted into this equation
;; is information pertaining to all of the checkpoints nodes within
;; the circuit. For each specific checkpoint a new equation is
;; added to the Boolean equation to form a new equation. As each
;; checkpoint equation is added, the equation is reduced and
;; simplified. After internal nodes are eliminated from the
;; equation, the sum of products equation is simplified and
;; returned.
;; -- A stepwise synopsis of this process is given at the start of
;; this file.

(define (generate-equation intermediate-format)

  (writeln "Processing....")
  (let* ( ; REDUCTION TO A SINGLE EQUATION
         ; make lists of internal, input nodes which do not fanout,
         ; input, and output nodes
         (internal-nodes (get-internal-nodes intermediate-format)))

```

```

(input-nodes-lf (get-input-nodes-less-fanouts
                           intermediate-format))
  (input-nodes    (get-input-nodes intermediate-format))
  (output-nodes   (get-output-nodes intermediate-format))
; give fanout nodes unique identifiers for each branch
  (prefix-list   (make-unique-fanouts intermediate-format))
; convert to a single equation (sum of products, f=0, form)
  (sopform      (make-sop prefix-list))
; convert input nodes which do not fanout to a new format
; e.g. B-- ==> BX-
  (sopform-1     (convert-input-nodes sopform
                                         input-nodes-lf))
; make a list of all the nodes which are fanout nodes
  (fanout-nodes  (get-unique-fanouts sopform-1))
; insert checkpoint equations for fanouts
  (sopform-2     (insert-checkpoint-eqns fanout-nodes
                                         sopform-1))
; make a list of the input nodes
  (input-nodes-lf-1 (convert-input-nodes input-nodes-lf
                                         input-nodes-lf))
; insert ckpt eqns for input nodes which don't fanout
  (sopform-3     (insert-checkpoint-eqns input-nodes-lf-1
                                         sopform-2))
; elim internal nodes (keep inputs, output, cp variables)
  (sopform-4     (eliminate sopform-3 internal-nodes))
; make list of the ckpt vars of the form AX0, BO0, etc.
; and build the constraint equation from this list
  (cp-variables  (get-cp-variables sopform-4))
  (constraint-eqn (make-constraint-equation cp-variables))
  (checkpt-vars   (flatten constraint-eqn))
; construct the single Boolean equation
  (output-eqn    (simplify (add sopform-4 constraint-eqn)))
; create the output list
  (output (cons output-eqn
                (cons input-nodes
                      (cons output-nodes
                            (list checkpt-vars)))))) )
output )

;; (GET-INTERNAL-NODES prefix-list)
;;
;; Parameters:
;;   prefix-list - a list of the form: ((eq ----- -----)
;;                                         (le ----- -----)
;;                                         :         ))
;;
;; -- GET-INTERNAL-NODES works by getting first all of the nodes in
;;   the circuit and subtracting the input and the output nodes.
;; -- GET-ALL-NODES returns all of the nodes in the circuit.

```

```

;; -- GET-INPUT-NODES returns the input nodes of the circuit.
;;   GET-SUBLIST subtracts the input nodes from all of the nodes
;;   leaving the internal nodes and output nodes
;;   (NODES-LESS-INPUT-NODES).
;; -- GET-OUTPUT-NODES returns the output nodes of the circuit.
;;   GET-SUBLIST subtracts the output nodes from the
;;   NODES-LESS-INPUT-NODES leaving the INTERNAL-NODES.

(define (get-internal-nodes prefix-list)
  (let* ( (all-nodes (get-all-nodes prefix-list))
          (nodes-less-input-nodes
            (get-sublist all-nodes
                         (get-input-nodes prefix-list)))
          (internal-nodes
            (get-sublist nodes-less-input-nodes
                         (get-output-nodes prefix-list))) )
    internal-nodes))

;; (GET-ALL-NODES prefix-list)
;;
;; Parameters:
;;   prefix-list - a list of the form: ((eq ----- -----)
;;                                     (le ----- -----)
;;                                     :      )
;;
;; -- GET-ALL-NODES calls GET-NODES which returns a list of all of the
;; nodes in the circuit. Since GET-NODES does not remove
;; duplicates of nodes, REMOVE-DUPLICATES is called to remove
;; duplicates in the list.

(define (get-all-nodes prefix-list)
  (remove-duplicates (get-nodes prefix-list)))

;; (GET-INPUT-NODES prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                     (le ----- -----)
;;                                     :      )
;;
;; -- Accepts a list in prefix form and returns a list of the nodes
;; which are outputs for the given system of equations. The
;; equations represent a combinational circuit.
;; -- Input nodes are all nodes which occur only on the right hand
;; side of the system of equations.
;; -- GET-INPUT-NODES takes each equation, and determines the symbols
;; on the left hand side by calling GET-NODES-ON-LEFT.

```

```

;; GET-NODES-ON-RIGHT returns the nodes on the right hand side.
;; The nodes on the left are then subtracted from the nodes on the
;; right (using GET-SUBLIST) yielding the input nodes. A list of
;; the input nodes is returned.

(define (get-input-nodes prefix-list)
  (let ( (nodes-on-left (remove-duplicates
                         (get-nodes-on-left prefix-list)))
        (nodes-on-right (remove-duplicates
                         (get-nodes-on-right prefix-list))) )
    (get-sublist nodes-on-right nodes-on-left)))

;; (GET-OUTPUT-NODES prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                         (le ----- -----)
;;                                         :      )
;;
;; -- Accepts a list in prefix form and returns a list of the nodes
;; which are outputs for the given system of equations. The
;; equations represent a combinational circuit.
;; -- Output nodes are all nodes which occur only on the left hand
;; side of the system of equations.
;; -- GET-OUTPUT-NODES takes the equations, and determines the symbols
;; on the left hand side by calling GET-NODES-ON-LEFT.
;; GET-NODES-ON-RIGHT determines the nodes on the right hand side.
;; The nodes on the right are then subtracted from the nodes on the
;; left (using GET-SUBLIST) yielding the output nodes. A list of
;; the output nodes is returned.

(define (get-output-nodes prefix-list)
  (let ( (nodes-on-left (remove-duplicates
                         (get-nodes-on-left prefix-list)))
        (nodes-on-right (remove-duplicates
                         (get-nodes-on-right prefix-list))) )
    (get-sublist nodes-on-left nodes-on-right)))

;; (GET-SUBLIST list-1 list-2)
;;
;; Parameters:
;;   list-1 -- an arbitrary list
;;   list-2 -- an arbitrary list
;;
;; -- GET-SUBLIST takes two lists and returns the items in list-1 that
;; are not members of list-2.
;; -- Duplicates are removed from the returned list.

```

```

(define (get-sublist list-1 list-2)
  (cond ( (null? list-1) '() )
        ; the first element of list-1 is an element of list-2
        ( (member (car list-1) list-2)
          (get-sublist (cdr list-1) list-2) )
        ; the first element of list-1 is not an element of list-2
        ( else
          (remove-duplicates
            (cons (car list-1)
                  (get-sublist (cdr list-1) list-2))))))

;; (GET-NODES-ON-RIGHT prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                         (le ----- -----)
;;                                         : -----)
;;
;; -- GET-NODES-ON-RIGHT gets nodes on the right side of the
;;   equations.
;; -- GET-NODES is used to get the nodes from the right hand side of
;;   the prefix-list. A list of nodes is returned.
;; -- Note: Duplicates are NOT removed from the list.

(define (get-nodes-on-right prefix-list)
  (if (null? prefix-list)
      '()
      (append (get-nodes (caddar prefix-list))
              (get-nodes-on-right (cdr prefix-list)))))

;; (GET-NODES-ON-LEFT prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                         (le ----- -----)
;;                                         : -----)
;;
;; -- GET-NODES-ON-LEFT gets nodes on the left side of the equation.
;; -- GET-NODES is used to get the nodes from the left hand side of
;;   the prefix-list. A list of nodes is returned.
;; -- Note: Duplicates are NOT removed from the list.

(define (get-nodes-on-left prefix-list)
  (if (null? prefix-list)
      '()
      (append (get-nodes (cadar prefix-list))

```

```

        (get-nodes-on-left (cdr prefix-list)))))

;; (GET-NODES lst)
;;
;; Parameters:
;;   lst -- a list in prefix form, i.e., (+ (* A B) (NOT C))
;;
;; -- GET-NODES accepts a list in prefix form and returns a list of
;;   all of the symbols in the list which are atoms, but are not
;;   token symbols.
;; -- TOKEN-SYMBOL? is used to determine if an atom is a token symbol.
;; -- GET-NODES extracts atoms which are included in nested lists.

(define (get-nodes lst)
  (cond ( (null? lst) '())
        ; if the list is atomic and not a token symbol,
        ; then return it in a list
        ( (and (atom? lst)
               (not (token-symbol? lst)))
          (list lst) )
        ; if the list is atomic and a token symbol, return nil
        ( (and (atom? lst)
               (token-symbol? lst))
          '() )
        ; otherwise, break apart the list
        ( else
          (let ((first-symbol (car lst))
                (rest-of-list (cdr lst)))
            ; if the first symbol is an atom, determine
            ; the type of symbol--if it is a token symbol,
            ; ignore it; if it is not, then add it to
            ; returned list--make a recursive call either way
            (cond ( (atom? first-symbol)
                    (if (not (token-symbol? first-symbol))
                        (cons first-symbol
                              (get-nodes rest-of-list))
                        (get-nodes rest-of-list)) )
                  ; otherwise, make recursive calls
                  ( else
                    (append (get-nodes first-symbol)
                            (get-nodes rest-of-list)) ))))) )

;; (GET-INPUT-NODES-LESS-FANOUTS prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                         (le ----- -----))

```

```

;;
;:      )

;;
;; -- GET-INPUT-NODES-LESS-FANOUTS returns the input nodes that do not
;;     fanout.  GET-INPUT-NODES returns a list of INPUT-NODES.
;; -- GET-FANOUT-NODES returns the FANOUT-NODES of the circuit.
;; -- GET-SUBLIST subtracts the input nodes which fanout from the
;;     complete list of input nodes, returning the inputs that do not
;;     fanout.

(define (get-input-nodes-less-fanouts prefix-list)
  (let ( (input-nodes (get-input-nodes prefix-list))
         (fanout-nodes (get-fanout-nodes prefix-list)) )
    (get-sublist input-nodes fanout-nodes)))

;;
;; (GET-FANOUT-NODES prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                         (le ----- -----)
;;                                         :       )
;;
;; -- Accepts a list in prefix form and returns a list of the nodes
;; which fanout for the given system of equations.  The equations
;; represent a combinational circuit.
;; -- The nodes which fanout are those nodes which occur in multiples
;; on the right hand side of the system of equations.
;; -- That is, take each equation,
;;   (eq left-hand-side right-hand-side), and determine the symbols
;; on the right-hand-side.  Then compile a list of symbols which
;; are the symbols from all of the right-hand-sides put together.
;; The fanouts are the symbols which occur more than once.
;; -- GET-NODES-ON-RIGHT returns the sum of the symbols from the right
;; hand side from the prefix-list.
;; -- GET-MULTIPLE-OCCURRENCES determines the symbols which occur more
;; than once from a given input list, and returns a list of those
;; symbols.

(define (get-fanout-nodes prefix-list)
  (let ( (nodes-on-right (get-nodes-on-right prefix-list)) )
    (get-multiple-occurrences nodes-on-right)))

```

```

;; (GET-MULTIPLE-OCCURRENCES input-list)
;;
;; Parameters:
;;   input-list -- an arbitrary list
;;
;; -- GET-MULTIPLE-OCCURRENCES determines the symbols which occur more
;;   than once from a given input list, and returns a list of those
;;   symbols.
;; -- Duplicates are removed from the list which is returned.

(define (get-multiple-occurrences input-list)
  (cond ( (null? input-list) '() )
        ; the car of input-list occurs at least once
        ; in the remainder of the list
        ( (member (car input-list) (cdr input-list))
          (remove-duplicates
            (cons (car input-list)
                  (get-multiple-occurrences (cdr input-list))))) )
        ; the car of the input-lst was not an elt of the rest of list
        ( else
          (get-multiple-occurrences (cdr input-list)) )))

;;
;; (REMOVE-DUPLICATES lst)
;;
;; Parameters:
;;   lst -- an arbitrary list
;;
;; -- REMOVE-DUPLICATES removes duplicates from the first level
;;   of the input list.

(define (remove-duplicates lst)
  (cond ( (null? lst)
           '() )
        ( (member (car lst) (cdr lst))
          (remove-duplicates (cdr lst)) )
        ( else
          (cons (car lst) (remove-duplicates (cdr lst))))))

;;
;; (MAKE-UNIQUE-FANOUTS prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
;;                                         (le ----- -----)
;;                                         :      )
;;
;; -- MAKE-UNIQUE-FANOUTS accepts a prefix list and modifies the

```

```

;; symbols so that a node which fans out has each fanout branch
;; replaced by a unique symbol.
;; -- For example, if B-- is a node which fans out, and the following
;; list is the input list:
;;      ((EQ E-- (NOT (* A-- B--)))
;;       (EQ F-- (NOT B--))
;;       (EQ Z-- (NOT (* E-- F--))))
;; MAKE-UNIQUE-NODES will return the following list:
;;      ((EQ E-- (NOT (* A-- B0-)))
;;       (EQ F-- (NOT B1-))
;;       (EQ Z-- (NOT (* E-- F--))))
;; -- GET-FANOUT-NODES is called to get the FANOUT-NODES.
;; MAKE-UNIQUE-FANOUTS-1 is then passed the PREFIX-LIST and the
;; FANOUT-NODES.

(define (make-unique-fanouts prefix-list)
  (let ( (fanout-nodes (get-fanout-nodes prefix-list)) )
    (make-unique-fanouts-1 fanout-nodes prefix-list)))

;; (MAKE-UNIQUE-FANOUTS-1 nodes prefix-list)
;;
;; Parameters:
;;   nodes - a list of the fanout nodes of the circuit
;;   prefix-list -- a list of the form:  ((eq _____ _____)
;;                                         (le _____ _____)
;;                                         :
;;                                         )
;;
;; -- MAKE-UNIQUE-FANOUTS-1 takes a node off the list of fanout nodes
;; and calls REPLACE-NODE which replaces a single node with unique
;; identifiers (right-hand side of the equation only).
;; -- MAKE-UNIQUE-FANOUTS-1 then takes the NEW-PREFIX-LIST and calls
;; itself recursively until all of the fanout nodes have been
;; replaced by unique identifiers.

(define (make-unique-fanouts-1 nodes prefix-list)
  (if (null? nodes)
      prefix-list
      (let ( (new-prefix-list
              (replace-node (car nodes) prefix-list 0)) )
        (make-unique-fanouts-1 (cdr nodes) new-prefix-list)))))

;; (REPLACE-NODES node-to-replace prefix-list fanout-number)
;;
;; Parameters:
;;   node-to-replace - the fanout node which will be replaced with
;;                     unique identifiers
;;   prefix-list - a list of the form:  ((eq _____ _____))

```

```

;;          (le _____)
;;          :         )
;; fanout-number - the number which will be used when replacing the
;;               next fanout branch
;;
;; -- REPLACE-NODE takes the prefix-list, and one line at a time looks
;; for the node to replace on the right hand side of the equation,
;; if it is there, then it is replaced.
;; -- REPLACE-NODE is then called recursively to replace successive
;; occurrences of the fanout node. FANOUT-NUMBER is incremented so
;; the next occurrence receives a higher number.
;; -- ON-RIGHT-SIDE? checks the RIGHT-SIDE of the first list in
;; PREFIX-LIST to see if the NODE-TO-REPLACE is in the list. If
;; the following list is the first list in PREFIX-LIST:
;;           (EQ E (NOT (* A-- B--))),
;; then RIGHT-SIDE would be the list: (NOT (* A-- B--)). (This is
;; returned by (caddr prefix-list)). If the NODE-TO-REPLACE is
;; B--, then ON-RIGHT-SIDE? would check to see if B-- is an element
;; of the RIGHT-SIDE.
;; -- Once it has been determined that NODE-TO-REPLACE is on the
;; RIGHT-SIDE of a given equation, then REP-NODE is called to
;; replace the fanout node with the unique identifier. The first
;; list in the PREFIX-LIST, the NODE-TO-REPLACE, and the
;; FANOUT-NUMBER are passed to REP-NODE.

(define (replace-node node-to-replace prefix-list fanout-number)
  (if (null? prefix-list)
      '()
      (let ( (right-side (caddr prefix-list)) )
        (if (on-right-side? node-to-replace right-side)
            (append (list (rep-node node-to-replace
                                      (car prefix-list)
                                      fanout-number))
                    (replace-node node-to-replace
                                  (cdr prefix-list)
                                  (1+ fanout-number)))
            (append (list (car prefix-list))
                    (replace-node node-to-replace
                                  (cdr prefix-list)
                                  fanout-number))))))

;;
;; (REP-NODE node-to-replace prefix-list number)
;;
;; Parameters:
;;   node-to-replace - the node to be replaced with a unique
;;                     identifier
;;   prefix-list - a list of the form: (EQ D-- (NOT (* A-- B--)))
;;   number - the number to used to form the unique identifier
;;

```

```

;; -- REP-NODE accepts a prefix-list of and replaces the
;;   NODE-TO-REPLACE with a numbered version of that node.
;; -- If B-- is the NODE-TO-REPLACE, and the PREFIX-LIST is the
;;   example above, then REP-NODE will return
;;   (EQ D-- (NOT (* A-- B0-))) assuming that NUMBER is 0.
;; -- REP-NODE disassembles the list into a HEAD and TAIL.  REP-NODE-1
;;   is then called to make the substitution for the NODE-TO-REPLACE.

(define (rep-node node-to-replace prefix-list number)
  (let* ( (tail (caddr prefix-list))
          (head (get-sublist prefix-list (list tail)))
          (new-tail (rep-node-1 node-to-replace tail number)) )
    (append head (list new-tail)))))

;; (REP-NODE-1 node-to-replace right-side number)
;;
;; Parameters:
;;   node-to-replace - the node to be replaced with a unique
;;                     identifier
;;   right-side - the right-side of a given equation,
;;                e.g. (NOT (* A-- B--))
;;   number - the number to be used to form the unique identifier
;;
;; -- REP-NODE-1 is a helping procedure for REP-NODE.  REP-NODE-1
;; accepts the tail part of the prefix-list from REP-NODE
;; (RIGHT-SIDE).
;; -- PUT-IN-NUMBER is the procedure which actually makes the
;; replacement for the symbol.

(define (rep-node-1 node-to-replace right-side number)
  ; the right-side is nil
  (cond ( (null? right-side) '() )
        ; the right-side is actually a symbol
        ( (symbol? right-side)
            (if (equal? right-side node-to-replace)
                (put-in-number right-side number)
                right-side))
        ; the head of the right-side is a list
        ( (list? (car right-side))
            (append (list (rep-node-1 node-to-replace
                                       (car right-side) number))
                    (rep-node-1 node-to-replace
                               (cdr right-side) number)) )
        ; the head of the right-side is a symbol
        ( else
            (if (equal? node-to-replace (car right-side))
                (cons (put-in-number (car right-side) number)
                      (cdr right-side))
                (cons (car right-side)

```

```

        (rep-node-1 node-to-replace
                      (cdr right-side) number)))))

;; (PUT-IN-NUMBER symbol number)
;;
;; Parameters:
;;   symbol - the symbol that will be replaced by the unique
;;             identifier
;;   number - the number to be used to form the unique identifier
;;
;; -- PUT-IN-NUMBER accepts a SYMBOL of the form B-- and a NUMBER and
;; returns a symbol of the form B0- where 0 is NUMBER.
;; -- First the symbol is converted to a list. PUT-IN-NUMBER-1 is
;; passed the LIST-FORM and NUMBER and makes the appropriate
;; substitution in the list. Then, the list is converted back to a
;; symbol which is returned.

(define (put-in-number symbol number)
  (let* ( (string-form (symbol->string symbol))
          (list-form (string->list string-form)) )
    (string->symbol (list->string (put-in-number-1 list-form
                                                       number))))))

;; (PUT-IN-NUMBER-1 list-form number)
;;
;; Parameters:
;;   list-form - a list of characters
;;   number - the number used to form the new list
;;
;; -- PUT-IN-NUMBER-1 is a helping procedure for PUT-IN-NUMBER. It
;; accepts a list of the characters that comprise the symbol input
;; to PUT-IN-NUMBER.
;; -- PUT-IN-NUMBER-1 then scans this list until the last two
;; characters, (#\--#\--) are reached. The number input to
;; PUT-IN-NUMBER-1 is substituted for the first #\-- character.

(define (put-in-number-1 list-form number)
  (if (equal? list-form '(#\--#\--))
      (append (string->list (number->string number '(int)))
              '#\--)
      (cons (car list-form)
            (put-in-number-1 (cdr list-form) number)))))

;; (ON-RIGHT-SIDE? node right-side)
;;

```

```

;; Parameters:
;;   node - a node in the circuit
;;   right-side - the right side of an equation
;;
;; -- ON-RIGHT-SIDE? is a predicate procedure called by REPLACE-NODE
;; to determine if a given node is on the right-side of an equation
;; in prefix-form.
;; -- ON-RIGHT-SIDE? is called recursively until the NODE can be
;; tested for equality against every symbol on the RIGHT-SIDE.

(define (on-right-side? node right-side)
  ; the right-side is nil
  (cond ( (null? right-side) '() )
        ; the right-side is a symbol
        ( (symbol? right-side)
          (if (eq? node right-side)
              #T
              '() ) )
        ; the head of the right-side is a list
        ( (list? (car right-side))
          (or (on-right-side? node (car right-side))
              (on-right-side? node (cdr right-side)))) )
        ; the head of the right-side is a symbol
        ( else
          (if (eq? node (car right-side))
              #T
              (on-right-side? node (cdr right-side)))) )

;; (MAKE-SOP lst)
;;
;; Parameters:
;;   lst -- a list in prefix-form:
;;
;;   ( (EQ (* (NOT A--) (* (NOT B--) (NOT C--)))
;;         (* (NOT X--) (* Y-- Z--)))
;;         (EQ (+ (* A-- B--) (* A-- C--))
;;             (+ (* (NOT X--) (NOT Y--)) (* (NOT Y--) (NOT Z--))))) )
;;
;; -- MAKE-SOP is used to convert a list in prefix-form into an
;; equivalent Sum of Products (SOP) form. The SOP form is
;; characterized by the transformation of several equations into a
;; single equation of the form: f = 0. All information that was
;; included in the system of Boolean equations is available in this
;; single equation.
;; -- For example, for the system of equations above, the equation
;; returned would be:
;;
;;   ( (A-- (X--) Y-- Z--) ((A--) B-- (X--) Y-- Z--))
;;     ((A--) (B--) C-- (X--) Y-- Z--) ((A--) (B--) (C--) X--))

```

```

;; ((A--) (B--) (C--) (X--) Y-- (Z--)) (A-- B-- X-- Y--)
;; (A-- B-- X-- (Y--) Z--)(A-- B-- (X--) Y--)(A-- (B--) C-- X-- Y--)
;; (A-- (B--) C-- X-- (Y--) Z--) (A-- (B--) C-- (X--) Y--)
;; (A-- (B--) (C--) (Y--) (Z--)) (A-- (B--) (C--) (X--) (Y--))
;; ((A--) (Y--) (Z--)) ((A--) (X--) (Y--)) )

;;
;; where each element of the list is a term of the SOP form.
-- MAKE-SOP returns nil if the input lst is nil.
-- If the input parameter is atomic, then MAKE-SOP takes the atom
and wraps it appropriately in lists to conform to the SOP
format. For example, the symbol A in SOP format would be
represented as ((A)).
-- If the first element of lst is atomic, then it must be a formula
in prefix form. Then check to determine the first element.
-- If it is NOT, then the SOP form of the second element must be
complemented. COMPLEMENT takes an SOP list and returns the
complement of the list in SOP form.
-- If the first element is a valid Boolean operator, then the
operation must be performed on the SOP forms of the second
and third elements. MAKE-SOP is called recursively to make
the second and third elements into SOP form. ADD, MULT, and
XOR are procedures which add, multiply, and take the
exclusive-or, respectively, of two lists in SOP form and
return the result in SOP form.
-- Otherwise, the input list is a list of lists - assume that these
lists are either a system of equations in prefix form, or they
are a list of formulas in prefix form with an implied OR.
MAKE-SOP breaks these lists up accordingly, and makes them into
SOP forms.
-- This is done one line at a time in which the first list of
lst is made into SOP form.
-- If the first element of this first list is EQ, then the
second and third elements of this list must be exclusive-ORed
together to get the form f = 0. Procedure XOR is used to
perform this operation. ADD is then used to add the SOP form
of the first list to the SOP form of the rest-of-list. A
recursive call is used to obtain the SOP form of
rest-of-list.
-- If the first element of the first list is LE, then the SOP
form of the second element must be MULTIplied by the
COMPLEMENT of the SOP form of the third element. The result
is ADDED to the SOP form of the rest of the list. Again,
this is done to realize the form: f = 0. Similar operations
are done when the first element is GE, however, the second
and third element are reversed from the LE case.
-- Otherwise, assume that each sublist of lst is a formula and
ADD its SOP form to the SOP form of the rest-of-list

(define (make-sop lst)
  (cond ( (null? lst) '())
        ; if lst is atomic return it in SOP list format

```

```

( (atom? lst)
  (list (list lst)) )
; if the first element is atomic, then lst is in prefix form
( (atom? (car lst))
  (let ((first-elt (car lst))
        (second-elt (cadr lst)))
    (if (eq? 'NOT first-elt)
        ; if first-elt is NOT, then complement the SOP form
        ; of the second element
        (if (atom? second-elt)
            (list (list (list second-elt)))
            (complement (make-sop second-elt))))
        ; if the first-elt is a valid Boolean operator, then
        ; perform the operation on the SOP forms of the
        ; second and third elements
        (cond ( (or (eq? '+ first-elt) (eq? 'OR first-elt))
                (add (make-sop second-elt)
                     (make-sop (caddr lst))))
              ( (or (eq? '* first-elt) (eq? 'AND first-elt))
                (mult (make-sop second-elt)
                      (make-sop (caddr lst))))
              ( (or (eq? '! first-elt) (eq? 'XOR first-elt))
                (xor (make-sop second-elt)
                     (make-sop (caddr lst))))
              ( else
                '() )))))
; the input lists is a list of lists - assume that these
; lists are a system of equations in prefix form, break up
; accordingly, and make into SOP forms
(else
  (let* ((first-list (car lst))
         (rest-of-list (cdr lst))
         (first-elt (car first-list))
         (second-elt (cadr first-list))
         (third-elt (caddr first-list)))
    (cond ( (eq? 'EQ first-elt)
            (add (xor (make-sop second-elt)
                      (make-sop third-elt))
                 (make-sop rest-of-list)))
          ; if first-elt of first-list is LE, then take
          ; MULT the SOP form of second-elt by the
          ; COMPLEMENT of the SOP form of third-elt.
          ; ADD the result to the SOP form of rest-of-list
          ( (eq? 'LE first-elt)
            (add (mult (make-sop second-elt)
                       (complement (make-sop third-elt)))
                 (make-sop rest-of-list)))
          ; if first-elt of first-list is LE, then take
          ; MULT the COMPLEMENT of the SOP form of
          ; second-elt by the SOP form of third-elt.
          ; ADD the result to the SOP form of rest-of-list

```

```
( (eq? 'GE first-elt)
  (add (mult (complement (make-sop second-elt))
             (make-sop third-elt))
         (make-sop rest-of-list)) )
; otherwise, assume that each sublist of lst is a
; formula and add its SOP form to the SOP form of
; the rest-of-list
(else
  (add (make-sop first-list)
    (make-sop rest-of-list)) ))))))
```

```

;;;;;;;;
;;                               Filename: eqn-gena.s
;;
;; Requires the following files to be loaded: new-sop.s, eqn-gen.s
;; NOTE: eqn-gena.s is a continuation of file eqn-gen.s
;;
;; See the header of file eqn-gen.s for a description of the
;; procedures contained in this file.
;;;;;;;

;; (CONVERT-INPUT-NODES sop-lst input-nodes)
;;
;; Parameters:
;;   sop-lst - an arbitrary list
;;   input-nodes - a list of the input nodes
;;
;; -- CONVERT-INPUT-NODES converts the nodes in the SOP-LST that are
;; members of the INPUT-NODES list to the form AX-.
;; -- If the following list is the SOP-LST:
;;
;;   ((E-- A-- B0-) ((E--) A-- (B0-)) ((E--) (A--)) (F-- B1-)
;;    ((F--) (B1-)) (Z-- E-- F--) ((Z--) E-- (F--)) ((Z--) (E--)))
;;
;; and (A--) is the list of INPUT-NODES, then CONVERT-INPUT-NODES
;; would return the following list:
;;
;;   ((E-- AX- B0-) ((E--) AX- (B0-)) ((E--) (AX-)) (F-- B1-)
;;    ((F--) (B1-)) (Z-- E-- F--) ((Z--) E-- (F--)) ((Z--) (E--)))
;;
;; -- CONVERT-INPUT-NODES-1 is called to convert a single node from
;; the INPUT-NODES returning a NEW-SOP-LST. CONVERT-NODES then
;; calls itself recursively until all of the nodes from the
;; INPUT-NODES list have been converted.

(define (convert-input-nodes sop-lst input-nodes)
  (if (null? input-nodes) .
      sop-lst
      (let ( (new-sop-lst
              (convert-input-nodes-1 (car input-nodes) sop-lst)) )
        (convert-input-nodes new-sop-lst (cdr input-nodes)) )))

;;
;; (CONVERT-INPUT-NODES-1 node sop-lst)
;;
;; Parameters:
;;   node - the node to be converted
;;   sop-lst - an equation in sum of products list form
;;
;; -- CONVERT-INPUT-NODES-1 breaks down the SOP-LST until it finds the

```

```

;; node to be converted. Once this node is found,
;; CONVERT-INPUT-NODE is called to actually convert the node.
;; -- After the SOP-LST is decomposed and the appropriate nodes are
;; converted the list is reassembled and returned.

(define (convert-input-nodes-1 node sop-lst)
    ; the sop-lst is nil
  (cond ( (null? sop-lst) '() )
        ; sop-lst is a symbol
        ( (symbol? sop-lst)
            (if (equal? sop-lst node)
                (convert-input-node node)
                sop-lst) )
        ; the first element of sop-lst is a symbol
        ( (symbol? (car sop-lst))
            (if (equal? (car sop-lst) node)
                (cons (convert-input-node node)
                      (convert-input-nodes-1 node (cdr sop-lst)))
                (cons (car sop-lst)
                      (convert-input-nodes-1 node (cdr sop-lst)))) )
        ; otherwise, the first element of sop-lst is a list
        ( else
            (append (list (convert-input-nodes-1 node (car sop-lst)))
                    (convert-input-nodes-1 node (cdr sop-lst)))))

;; (CONVERT-INPUT-NODE node)
;;
;; Parameter:
;;   node - a symbol of the form: A--
;;
;; -- CONVERT-INPUT-NODE converts the input NODE to a list of
;;   characters. The last two characters are then removed from the
;;   list and replaced by a "X--" suffix.

(define (convert-input-node node)
  (let* ( (node-1 (string->list (symbol->string node)))
         (symbol-less-suffix (remove-suffix node-1))
         (new-suffix (list #\X #\--))
         (new-symbol
           (string->symbol
             (list->string (append symbol-less-suffix
                                   new-suffix)))) )
    new-symbol))

;; (REMOVE-SUFFIX lst)
;;
;; Parameters:

```

```

;;      lst - an arbitrary list
;;
;; -- REMOVE-SUFFIX accepts a list, LST, and returns a list with its
;;      last two elements removed. It is assumed that LST has at least
;;      two elements in it.

(define (remove-suffix lst)
  (if (equal? (length lst) 2)
      '()
      (cons (car lst) (remove-suffix (cdr lst)))))

;;
;; (GET-UNIQUE-FANOUTS sop-lst)
;;
;; Parameters:
;;      sop-lst - an equation in sum of products list form
;;
;; -- GET-UNIQUE-FANOUTS returns a list of each of the fanout branches
;;      in the circuit. MAKE-UNIQUE-FANOUTS made each of the fanout
;;      branches into a unique node with a unique identifier. These
;;      identifiers had a number associated with them in the next to
;;      last digit.
;; -- GET-UNIQUE-FANOUTS scans the SOP-LST and builds a list of all of
;;      the symbols which have a number in the next to last alphanumeric
;;      digit.
;; -- GET-UNIQUE-FANOUTS-1 is a helping procedure which builds an
;;      initial list. REMOVE-DUPLICATES is called to remove duplicates
;;      from the initial list. This modified list is returned.
;; -- If the following list were input to GET-UNIQUE-FANOUTS:
;;
;;      ((E-- AX- B0-) ((E--) AX- (B0-)) ((E--) (AX-)) (F-- B1-)
;;       ((F--) (B1-)) (Z-- E-- F--) ((Z--) E-- (F--)) ((Z--) (E--)))
;;
;;      the list (B0- B1-) would be returned.

(define (get-unique-fanouts sop-lst)
  (remove-duplicates (get-unique-fanouts-1 sop-lst)))

;;
;; (GET-UNIQUE-FANOUTS-1 sop-lst)
;;
;; Parameters:
;;      sop-lst - an equation in sum of products list form
;;
;; -- GET-UNIQUE-FANOUTS-1 breaks down the SOP-LST to the symbol-level
;;      and examines each symbol to determine if it is a unique fanout
;;      node. UNIQUE-CP-NODE? is a predicate called to make this
;;      determination.
;; -- GET-UNIQUE-FANOUTS-1 calls itself recursively if a list is

```

```

;; encountered, breaking it up until a symbol is found and tested.
;; -- Duplicate occurrences of a given fanout node occur in the
;; SOP-LST. However, these are not removed by GET-UNIQUE-FANOUTS-1.

(define (get-unique-fanouts-1 sop-lst)
    ; sop-lst is nil
  (cond ( (null? sop-lst) '())
        ; the sop-lst is actually a symbol
        ( (symbol? sop-lst)
            (if (unique-cp-node? sop-lst)
                sop-lst
                '() ))
        ; the first element of sop-lst is a symbol
        ( (symbol? (car sop-lst))
            (if (unique-cp-node? (car sop-lst))
                (cons (car sop-lst)
                      (get-unique-fanouts-1 (cdr sop-lst)))
                (get-unique-fanouts-1 (cdr sop-lst)) ) )
        ; otherwise, the first element of the sop-lst is a list
        ( else
            (append (get-unique-fanouts-1 (car sop-lst))
                    (get-unique-fanouts-1 (cdr sop-lst))))))

;; (UNIQUE-CP-NODE? node)
;;
;; Parameter:
;;   node - an arbitrary node in the input circuit
;;
;; -- UNIQUE-CP-NODE? decomposes the NODE symbol into a list of
;; characters. GET-NEXT-TO-LAST-ELT is called to extract the next
;; to last character from this list.
;; -- NUMBER-CHAR? is called to determine whether the
;;   NEXT-TO-LAST-CHAR is a character representing a number between 0
;;   and 9. If so, UNIQUE-CP-NODE? returns #T, otherwise nil.

(define (unique-cp-node? node)
  (let* ( (node-1 (string->list (symbol->string node)))
         (next-to-last-char (get-next-to-last-elt node-1)) )
    (if (number-char? next-to-last-char)
        #T
        '() )))

;; (GET-NEXT-TO-LAST-ELT lst)
;;
;; Parameter:
;;   lst - an arbitrary list
;;

```

```

;; -- GET-NEXT-TO-LAST-ELT accepts an arbitrary list and returns the
;;     next to last element of the top level of the input LST.

(define (get-next-to-last-elt lst)
  (if (equal? (length lst) 2)
      (car lst)
      (get-next-to-last-elt (cdr lst)))))

;; (NUMBER-CHAR? symbol)
;;
;; Parameters:
;;   symbol - an arbitrary symbol
;;
;; -- NUMBER-CHAR? is a predicate procedure for determining whether
;;   a given symbol is actually a character representing a number
;;   between 0 and 9.

(define (number-char? symbol)
  (and (char<=? symbol #\9)
       (char>=? symbol #\0)) )

;;
;; (INSERT-CHECKPOINT-EQNS nodes sop-lst)
;;
;; Parameters:
;;   nodes - a list of the nodes in the circuit to be replaced by
;;           checkpoint equations
;;   sop-lst - an equation in sum of products list form
;;
;; -- INSERT-CHECKPOINT-EQNS adds a checkpoint equation for each of
;;   the NODES input to the procedure. This process is done
;;   iteratively whereby a checkpoint equation for each NODE is added
;;   to the current sum of products equation by the process of
;;   reduction to form a NEW-SOP-LST. Then, Boolean elimination is
;;   used to eliminate the symbol which represents the either the
;;   unique fanout branch or an input which does not fanout. Thus,
;;   after the appropriate equation has been inserted, the symbol
;;   representing the unique fanout branch or an input which does not
;;   fanout is removed from the equation, and the symbol representing
;;   the original node with two checkpoint variable symbols are
;;   added.
;;
;; -- INSERT-CHECKPOINT-EQNS-1 is the procedure called to perform this
;;   operation for each of the nodes. INSERT-CHECKPOINT-EQNS
;;   calls itself recursively until the list of nodes for which
;;   equations must be inserted is exhausted.

(define (insert-checkpoint-eqns fanout-nodes sop-lst)
  (if (null? fanout-nodes)

```

```

sop-lst
(let ( (new-sop-lst (insert-checkpoint-eqns-1
                           (car fanout-nodes) sop-lst)) )
  (insert-checkpoint-eqns (cdr fanout-nodes) new-sop-lst) )))

;; (INSERT-CHECKPOINT-EQNS-1 node sop-lst)
;;
;; Parameters:
;;   node - a single node to be replaced with a checkpoint equation
;;   sop-lst - an equation in sum of products list form
;;
;; -- INSERT-CHECKPOINT-EQNS-1 inserts the checkpoint equation for the
;;   NODE.
;; -- REPLACE-SYMBOL-WITH-EQN is call to produce the checkpoint
;;   equation for the given node. This equation is in f = 0 form.
;;   This allows the checkpoint equation to be added to the SOP-LST
;;   to form a NEW-SOP-LST.
;; -- The ECON procedure is used to conjunctively eliminate the NODE
;;   from the NEW-SOP-LST to produce an equation in which the symbol
;;   representing the unique fanout branch or the modified input are
;;   removed from the equation, and the symbol representing the
;;   original node plus two checkpoint variable symbols are added.
;;   (The original node was replaced with the unique symbol either by
;;   MAKE-UNIQUE-FANOUTS or CONVERT-INPUT-NODES.)
;; -- The list returned by ECON is then returned.

(define (insert-checkpoint-eqns-1 node sop-lst)
  (let ( (new-sop-lst (add sop-lst (replace-symbol-with-eqn node))) )
    (econ new-sop-lst (list node)) ))

;; (REPLACE-SYMBOL-WITH-EQN cp-symbol)
;;
;; Parameters:
;;   cp-symbol - the node symbol used to form the checkpoint equation
;;
;; -- REPLACE-SYMBOL-WITH-EQN returns a list representing the sum of
;;   products form of the checkpoint equation.
;; -- Mathematically, the unique node is represented by the equation:
;;     Unique-node = Original-node * CP-variable-0' + CP-variable-1
;; -- If the CP-SYMBOL input is A0-, then this equation would be:
;;     A0- = A-- * A00' + A01
;; -- This equation can be converted to f = 0 form to provide the
;;   Boolean sum of products representation. In this procedure, this
;;   form is returned instead of the equation above to eliminate
;;   processing of this conversion.
;; -- If the CP-SYMBOL input is A0-, then the following list would be
;;   returned by REPLACE-SYMBOL-WITH-EQN:

```

```

((AO- (A--) (AO1)) (AO- A00 (AO1)) ((AO-) AO1) ((AO-) A-- (AO0)))
;; -- This initial CP-SYMBOL is first converted to a list of
;;   characters. It two character suffix is removed by REMOVE-SUFFIX
;;   to provide the symbol on which to build the new suffixes (in the
;;   given example: A).
;; -- GET-FANOUT-NUMBER returns the number associated with the unique
;;   fanout branch. This number is used to build appropriate
;;   checkpoint variable symbols. In the given example, this number
;;   is 0. The checkpoint symbols A00 and A01 are built based on
;;   this number.
;; -- Once the appropriate variables have been assembled and converted
;;   to symbol form, then the output equation is built and returned.

(define (replace-symbol-with-eqn cp-symbol)
  (let* ( (cp-symbol-1 (string->list (symbol->string cp-symbol)))
         (symbol-less-suffix (remove-suffix cp-symbol-1))
         (fanout-number (get-next-to-last-elt cp-symbol-1))
         (suffix-1 (list #\` #\`))
         (suffix-2 (append (list fanout-number)
                           (string->list (number->string 0 '(int)))))
         (suffix-3 (append (list fanout-number)
                           (string->list (number->string 1 '(int)))))
         (symbol-1 (string->symbol
                           (list->string (append symbol-less-suffix
                                                 suffix-1))))
         (symbol-2 (string->symbol
                           (list->string (append symbol-less-suffix
                                                 suffix-2))))
         (symbol-3 (string->symbol
                           (list->string (append symbol-less-suffix
                                                 suffix-3)))))
        '(((,cp-symbol (,symbol-1) (,symbol-3))
          (,cp-symbol ,symbol-2 (,symbol-3))
          ((,cp-symbol) ,symbol-3)
          ((,cp-symbol) ,symbol-1 (,symbol-2)))) )

;; (ELIMINATE f nodes)
;;
;; Parameters:
;;   f - a list representing a Boolean equation in sum of products
;;       (f=0) form
;;   nodes - a list of nodes to be eliminated
;;
;; -- ELIMINATE breaks up the list of NODES and iteratively calls ECON
;;   to eliminate each of the nodes from the function F.
;; -- After a NEW-F is formed, ELIMINATE calls itself recursively
;;   until all nodes are eliminated.
;; -- It is assumed that NODES is a list of nodes

```

```

(define (eliminate f nodes)
  (if (null? nodes)
      f
      (let ( (new-f (econ f (list (car nodes))))) )
        (eliminate new-f (cdr nodes)))))

;; (GET-CP-VARIABLES sop-lst)
;;
;; Parameters:
;;   sop-lst - a list representing a Boolean equation in sum of
;;             products form
;;
;; -- GET-CP-VARIABLES accepts a list representing a Boolean equation
;; and returns a list representing one of the two checkpoint
;; variables for each checkpoint in the circuit. In this case,
;; that is all symbols in the SOP-LST that have a 0 in the last
;; alphanumeric position.
;; -- LAST-CHAR-EQ-ZERO? is a predicate procedure used to determine
;; whether a symbol has has a zero as its last character.
;; GET-CP-VARIABLES calls itself recursively until all symbols have
;; been tested in the SOP-LST.
;; -- REMOVE-DUPLICATES removes duplicate symbols from the list that
;; is created.

(define (get-cp-variables sop-lst)
  ; sop-lst is nil
  (cond ( (null? sop-lst) '() )
        ; sop-lst is a symbol
        ( (symbol? sop-lst)
            (if (last-char-eq-zero? sop-lst)
                sop-lst
                '() ) )
        ; the first element of the sop-lst is a symbol
        ( (symbol? (car sop-lst))
            (if (last-char-eq-zero? (car sop-lst))
                (remove-duplicates
                  (cons (car sop-lst)
                        (get-cp-variables (cdr sop-lst)))) )
                (remove-duplicates (get-cp-variables (cdr sop-lst)))) ) )
        ; the first element of the sop-lst is a list
        ( else
            (remove-duplicates
              (append (get-cp-variables (car sop-lst))
                    (get-cp-variables (cdr sop-lst)))) )))

;; (LAST-CHAR-EQ-ZERO? symbol)
;;

```

```

;; Parameter:
;;   symbol - an arbitrary symbol
;;
;; -- LAST-CHAR-EQ-ZERO? is a predicate for determining whether the
;;   last character of the SYMBOL is #\0.
;; -- The SYMBOL is first converted to a list of characters.
;;   GET-LAST-ELT returns the last element of this list of
;;   characters.

(define (last-char-eq-zero? symbol)
  (let* ( (symbol-1 (string->list (symbol->string symbol)))
         (last-char (get-last-elt symbol-1)) )
    (if (equal? last-char '#\0)
        #T
        '() )))

;;
;; (GET-LAST-ELT lst)
;;
;; Parameter:
;;   lst - an arbitrary list
;;
;; -- GET-LAST-ELT accepts an arbitrary list and returns the last
;;   element of the list.

(define (get-last-elt lst)
  (if (equal? (length lst) 1)
      lst
      (get-last-elt (cdr lst)))))

;;
;; (MAKE-CONSTRAINT-EQUATION lst)
;;
;; Parameters:
;;   lst - a list of checkpoint variables as constructed by
;;         GET-CP-VARIABLES
;;
;; -- MAKE-CONSTRAINT-EQUATION accepts the list of checkpoint
;;   variables and calls MAKE-CONSTRAINT-EQUATION-1 which makes an
;;   equation for each of the variables in the LST.
;; -- If the following list were input:
;;   (B00 B10 AX0)
;;   MAKE-CONSTRAINT-EQUATION would return:
;;   ((B00 B01) (B10 B11) (AX0 AX1))
;; -- MAKE-CONSTRAINT-EQUATION-1 would be passed the variables B00,
;;   B10, and AX0 in successive calls; returning in succession
;;   (B00 B01), (B10 B11), and (AX0 AX1).  MAKE-CONSTRAINT-EQUATION
;;   assembles these sublists to form the larger list which is
;;   returned.

```

```

(define (make-constraint-equation lst)
  (if (null? lst)
      '()
      (append (list (make-constraint-equation-1 (car lst)))
              (make-constraint-equation (cdr lst)))))

;; (MAKE-CONSTRAINT-EQUATION-1 symbol)
;;
;; Parameters:
;;   symbol - a symbol whose last character is a zero
;;
;; -- MAKE-CONSTRAINT-EQUATION-1 converts the SYMBOL to a list
;;   (SYMBOL-L) and calls DROP-LAST-CHAR to remove the last character
;;   from the list.
;; -- Then, a #\1 character is added to SYMBOL-LESS-LAST-CHAR to form
;;   a list representing a new symbol (SYMBOL-L-1). This list is
;;   then converted to a symbol (SYMBOL-1).
;; -- The input SYMBOL and SYMBOL-1 are then enclosed in a list and
;;   returned.

(define (make-constraint-equation-1 symbol)
  (let* ( (symbol-1 (string->list (symbol->string symbol)))
          (symbol-less-last-char (drop-last-char symbol-1))
          (suffix (string->list (number->string 1 '(int))))
          (symbol-1-1 (append symbol-less-last-char suffix))
          (symbol-1 (string->symbol (list->string symbol-1-1))) )
    (append (list symbol) (list symbol-1))))

;; (DROP-LAST-CHAR symbol-1)
;;
;; Parameter:
;;   symbol-1 - a list of characters which represent a symbol
;;
;; - DROP-LAST-CHAR returns the input list less the last character.

(define (drop-last-char symbol-1)
  (if (equal? (length symbol-1) 1)
      '()
      (cons (car symbol-1) (drop-last-char (cdr symbol-1)))))

;; (GET-CHECKPOINT-MODES prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq ----- -----)
```

```

;;
;;          (le _____)
;;          :      )
;;
;; -- Accepts a list in prefix form and returns a list of the
;; checkpoints for the given system of equations. The equations
;; represent a combinational circuit.
;; -- The procedure gets the nodes which fan out as well as the input
;; nodes. All checkpoints are derived either from fanouts or input
;; nodes.
;; -- The fanout nodes list is appended to the input node list.
;; Duplicates are removed in case an input node is also a node
;; which has a fanout.

(define (get-checkpoint-nodes prefix-list)
  (let ( (fanout-nodes (get-fanout-nodes prefix-list))
        (input-nodes (get-input-nodes prefix-list)) )
    (remove-duplicates (append input-nodes fanout-nodes)))))

;;
;; (GET-INPUT-NODES-WHICH-FANOUT prefix-list)
;;
;; Parameters:
;;   prefix-list -- a list of the form: ((eq _____)
;;                                     (le _____)
;;                                     :      )
;;
;; -- GET-INPUT-NODES-WHICH-FANOUT returns a list of input nodes
;; that fanout.
;; -- GET-INPUT-NODES is called to make a list of INPUT-NODES of the
;; circuit.
;; -- GET-INPUT-NODES-LESS-FANOUTS returns a list of
;; INPUT-NODES-LESS-FANOUTS.
;; -- GET-SUBLIST subtracts INPUT-NODES-LESS-FANOUTS from the
;; INPUT-NODES to form a list of input node that do fanout. This
;; list is returned.

(define (get-input-nodes-which-fanout prefix-list)
  (let ( (input-nodes (get-input-nodes prefix-list))
        (input-nodes-less-fanouts
          (get-input-nodes-less-fanouts prefix-list)) )
    (get-sublist input-nodes input-nodes-less-fanouts)))

```

```

;;;;;;;;;;;;;;;;
;;           Filename: tester.s
;;
;; This module is the portion of the diagnostic system which
;; provides the mechanisms for conducting the input-output
;; experiment. It takes the single Boolean equation produced by
;; the Equation Generation Module of the system, lists of the
;; inputs, outputs, and checkpoints, and produces test vectors for
;; the given circuit. Once the result of the test is known, the
;; result is fed back to the system which used it to create new
;; information about the circuit. Tests are conducted until it has
;; been determined that further information cannot be gained from
;; input-output tests. At this point, an equation exists which
;; holds all of the information known about the circuit, including
;; the state of faults and the actual circuit function. This
;; equation is returned in a list with the number of tests that
;; were conducted.
;;
;; Requires the files: boolean.fsl, interp.fsl, interp-a.fsl
;;
;;;;;;;;;;;;;;;;
;;
;; (TESTER input)
;;
;; Parameters:
;;   input - The output list from the Equation Generation Module of
;;          the diagnostic system. It consists of sublists which
;;          are the system equation generated by GENERATE-EQUATION,
;;          the INPUTS, OUTPUTS, and the circuit CHECKPOINTS.
;;
;; -- TESTER decomposes the INPUT into its composite sublists and
;; takes the Blake Canonical Form of the equation to form a new
;; equation (NEW-EQN).
;;
;; -- A TEST-INPUT is generated from this NEW-EQN by MAKE-TEST-INPUT.
;; Then TESTER-1 is called to begin the iterative testing process.

(define (tester input)
  (let* ( (equation  (car input))
          (inputs    (cadr input))
          (outputs   (caddr input))
          (checkpoints (cadddr input))
          (new-eqn   (bcf equation))
          (test-input (make-test-input new-eqn inputs checkpoints
                                      outputs)) )
    (tester-1 new-eqn test-input inputs checkpoints outputs 0) ))

;;
;; (TESTER-1 equation test-inp inputs checkpoints outputs test-no)
;;
;; Parameters:

```

```

;; equation - The system equation generated by GENERATE-EQUATION.
;; test-inp - The first test input generated by the system.
;; inputs - A list of the inputs of the circuit.
;; checkpoints - A list of the checkpoints in the circuit.
;; outputs - A list of the outputs of the circuit.
;; test-no - The current test number. Initially, this is 1.
;;
;; -- TESTER-1 is a helping procedure for TESTER. However, it is the
;; module that supervises the input-output experiment.
;; If the TEST-IMP is null, then another test could not be
;; generated from the system EQUATION. At this time, a message is
;; output and the system EQUATION is returned in a list along with
;; the TEST-N0 which indicates the number of tests that occurred.
;; -- The TEST-IMP is generated prior to TESTER-1 being called. If it
;; is not null, then a test was generated. PRINT-SUGGESTED-INPUT
;; outputs the list representing the test vector in a user-readable
;; form.
;; -- The user is then prompted for the RESULT of the test. The
;; RESULT is combined with the TEST-IMP by MAKE-NEW-INFO to make
;; NEW-INFOmation which can be added to the EQUATION. The
;; combination of the EQUATION and NEW-INFO forms a NEW-EQN. DCF
;; is a procedure used to generate the Diagnostic Canonical Form
;; which is a form of the equation necessary to generate new test
;; vector inputs.

(define (tester-1 equation test-inp inputs checkpoints outputs
                     test-no)
  (cond ( (null? test-inp)
          (writeln "New information cannot be obtained.")
          (newline)
          (cons test-no equation) )
        ( else
          ; print out the suggested input in a user-readable format
          (print-suggested-input test-inp)
          (newline)
          ; for the first test, give the user instructions
          (if (equal? 0 test-no)
              (writeln "If the output was 0, type 0 and <rtn>, else
type 1 and <rtn>.")
              '())
          ; prompt for the result
          (display "Enter the Resulting Output (0 or 1) --> ")
          ; read in the result, generate new info from the test input
          ; and the result, and make a NEW-EQN which contains the old
          ; EQUATION plus new information derived from the test
          (let* ( (result (read-line))
                  (new-info (make-new-info test-inp result outputs))
                  (new-eqn (dcf (cons new-info equation)
                                inputs outputs)) )
            ; make a recursive call using the NEW-EQN, generating a
            ; new TEST-INPUT on the fly, increment the TEST-N0

```

```

(tester-1 new-eqn
  (make-test-input new-eqn inputs checkpoints
    outputs)
  inputs
  checkpoints
  outputs
  (1+ test-no))) )))

;; (PRINT-SUGGESTED-INPUT lst)
;;
;; Parameters:
;;   lst - A list of the form ((A--) B-- C--), where the subelements
;;         are literals representing the inputs to the circuit.
;;
;; -- PRINT-SUGGESTED-INPUT prints out a message and then calls
;;   PRINT-SUGGESTED-INPUT-1 which outputs each of the inputs
;;   individually.

(define (print-suggested-input lst)
  (writeln "The Suggested Input is: ")
  (newline)
  (print-suggested-input-1 lst) )

;;
;; (PRINT-SUGGESTED-INPUT-1 lst)
;;
;; Parameters:
;;   lst - A list of the form ((A--) B-- C--), where the subelements
;;         are literals representing the inputs to the circuit.
;;
;; -- PRINT-SUGGESTED-INPUT-1 prints out the suggested input in a
;;   user-readable format. The input LST is of the form ((A--) B--
;;   C--), where each symbol is an input to the circuit. If a
;;   literal is enclosed in a sublist, then it should be set to 0.
;;   Otherwise, if it exists in the top-level of the list, then it
;;   should be set to 1.
;; -- In each call to PRINT-SUGGESTED-INPUT-1, one of the suggested
;;   inputs is output. Recursive calls are made until all of the
;;   suggested inputs have been output.
;; -- CONVERT-NODE-BACK is called to eliminate the suffix from each
;;   symbol. The symbol is then of the form that was originally
;;   input to the system by the user.

(define (print-suggested-input-1 lst)
  (if (null? lst)
      '()
      (let ( (first-term (car lst))
            (rest       (cdr lst)) )

```

```

; if the first-term is a symbol, it should be set to 1
; otherwise, if in a sublist, it should be set to 0
(if (symbol? first-term)
    (begin
        (writeln "      " (convert-node-back first-term)
                  " = 1")
        (print-suggested-input-1 rest))
    (begin
        (writeln "      " (convert-node-back (car first-term))
                  " = 0")
        (print-suggested-input-1 rest)) )))

;; (MAKE-NEW-INFO test-input result outputs)
;;
;; Parameters:
;;   test-input - A list of the form ((A--) B-- C--) which was the
;;               test vector generated by TEST-INPUT.
;;   result - A string representing the result of the test; either
;;           "1" or "0".
;;   outputs - A list of the outputs of the circuit.
;;
;; -- MAKE-NEW-INFO combines the TEST-INPUT with the OUTPUTS to make
;; new information about the state of the circuit.
;; -- The new information is based on the mathematical model that:
;;   TEST-INPUT ==> OUTPUT
;;   Translated into Boolean Algebra, this would be modeled
;;   TEST-INPUT OUTPUT
;;   This is then converted to the form
;;   TEST-INPUT * OUTPUT' = 0
;;   Lists are built appropriately to implement this last equation.
;;   This list is then added to the old equation to form an updated
;;   equation.
;; -- As currently implemented, it is assumed that OUTPUTS is a list
;;   of a single element representing a single output of the circuit.

(define (make-new-info test-input result outputs)
  (newline)
  (newline)
  (writeln "Processing....")
  (newline)
  (cond ( (equal? result "1")
          (append test-input (list outputs)) )
        ( (equal? result "0")
          (append test-input outputs) )))

;;
;; (DCF equation inputs outputs)
;;

```

```

;; Parameters:
;;   equation - The new equation formed by adding the new information
;;             generated by an input-output test to the old system
;;             equation. This equation is in f=0 form.
;;   inputs - A list of the inputs of the circuit.
;;   outputs - A list of the outputs of the circuit.

;;
;; -- DCF generates the "Diagnostic Canonical Form" of the system
;;   equation.
;; -- First, the Blake Canonical Form (BCF) is taken of the input
;;   EQUATION. This generates all of the possible consensus terms
;;   from the EQUATION.
;; -- The aim of the Diagnostic Canonical Form is to get the equation
;;   into the following form:
;;   A(x,y) z' + B(x,y) z + G(y) = 0 where x represents the circuit
;;   inputs, z the circuit outputs,
;;   and y the checkpoint variables
;; -- However, after getting the Blake Canonical Form of this
;;   equation, it may be in the form:
;;   A(x,y) z' + B(x,y) z + H(x,y) = 0
;; -- The G(y) term is made up of the elements of H(x,y) which have
;;   had the input variables stripped, or SIFTed, off. This can be
;;   done because the input variables are not constrained due to
;;   independence. Thus, the checkpoint variables they are combined
;;   with to form a term must be identically equal to 0.
;; -- SIFT forms the terms in G(y) which are added to the input
;;   EQUATION. UNABSORB is then called to execute absorptions caused
;;   by these new terms.

```

```

(define (dcf equation inputs outputs)
  (unabsorb (sift (bcf equation) inputs outputs)) )

```

```

;; (SIFT equation inputs outputs)

;;
;; Parameters:
;;   equation - The new equation formed by adding the new information
;;             generated by an input-output test to the old system
;;             equation. This equation is in f=0 form.
;;   inputs - A list of the inputs of the circuit.
;;   outputs - A list of the outputs of the circuit.

;;
;; -- SIFT is a helping procedure for the DCF procedure. It generates
;;   the G(y) terms from the H(x,y) terms in the equations listed
;;   above.
;; -- COMMON-ARGS? is used to determine whether any OUTPUTS are in a
;;   given term of the EQUATION. If they are, then this term is
;;   simply ignored. If they are not, then the INPUTS are
;;   disjunctively eliminated from the term to yield a term that is
;;   composed only of checkpoint variables.

```

```

;; -- SIFT calls itself recursively until all terms of the input
;;     EQUATION have been checked and modified if appropriate.

(define (sift equation inputs outputs)
  (cond ( (null? equation)
           '() )
        ( (not (common-args? outputs (car equation)))
           (cons (car (edis (list (car equation)) inputs))
                 (sift (cdr equation) inputs outputs) )))
        ( else
           (cons (car equation)
                 (sift (cdr equation) inputs outputs) ))))

;; (MAKE-TEST-INPUT equation inputs checkpoints outputs)
;;
;; Parameters:
;;   equation - The new equation formed by adding the new information
;;             generated by an input-output test to the old system
;;             equation. This equation is in f=0 form.
;;   inputs - A list of the inputs of the circuit.
;;   outputs - A list of the outputs of the circuit.
;;   checkpoints - A list of the checkpoints of the circuit.
;;
;; -- MAKE-TEST-INPUT uses EQUATION, the CHECKPOINTS, and the OUTPUTS,
;; to generate a test vector input.
;; -- MAKE-INPUT-EQUATION is passed the EQUATION, CHECKPOINTS, and
;; OUTPUTS. Boolean elimination is used to remove the CHECKPOINTS
;; and OUTPUTS from the EQUATION to get an INPUT-EQUATION in f=0
;; format. Solving this equation yields an effective input that
;; will yield new information about the circuit.
;; -- Because it is difficult to solve an equation in f=0 format, i.e.
;; all terms must be set to 0, the INPUT-EQUATION is complemented
;; to get the f=1 form. Then, only a single TERM need be set to 1
;; to solve the equation. DISPLAY-CIRCUIT-FUNCTION-1 is called to
;; display the f=0 equation that must be solved.
;; -- When the INPUT-EQUATION becomes equal to 1, or in the
;; representation used in this system, '(()), then further
;; effective inputs cannot be generated. At this time '() is
;; returned.
;; -- All of the INPUTS may not exist as literals of TERM. COMBINE is
;; used to insert the INPUTS that are not literals of TERM into
;; term. Due to the nature of Boolean Algebra, these missing
;; literal can be arbitrarily set to 0 or 1. In this
;; implementation, the missing literals are set to 1. SORT-TERM is
;; called to generate a test vector in sorted order.

(define (make-test-input equation inputs checkpoints outputs)
  (let* ( (input-equation (make-input-equation equation checkpoints
                                                outputs)))

```

```

        (term (car (bcf (complement input-equation)))) )
(newline)
; if the input function was 0, then any input is effective
; i.e. there are no constraints on input variables that are
; required to yield new information about the circuit
(if (null? input-equation)
    (writeln "The Input Equation is: 0 = 0")
    (begin
        (display "The Input Equation is: ")
        (display-circuit-function-1 input-equation)
        (writeln "= 0")))
; if the input was 1 i.e. '(()), then return nil to signify that
; a new input function cannot be generated. Otherwise, take the
; term, fill in the missing literals, sort it alphabetical order,
; and return.
(if (equal? input-equation '(()))
    '()
    (sort-term (combine term inputs)))))

;; (MAKE-INPUT-EQUATION equation checkpoints outputs)
;;
;; Parameters:
;;   equation - The new equation formed by adding the new information
;;             generated by an input-output test to the old system
;;             equation. This equation is in f=0 form.
;;   checkpoints - A list of the checkpoints of the circuit.
;;   outputs - A list of the outputs of the circuit.
;;
;; -- MAKE-INPUT-EQUATION accepts an equation of the form:
;;    P(x,y,z) = 0 where x are the inputs of the circuit,
;;                y are the checkpoints of the circuit,
;;                and z the outputs of the circuit.
;; -- Conjunctive ELIMINATION is used to remove the checkpoints from
;; the equation. This leaves an equation of the following form:
;;    A(x) z' + B(x) z = 0
;; -- Disjunctive elimination, performed by EDIS, yields an equation
;; of the form:
;;    A(x) + B(x) = 0
;; -- The Blake Canonical Form of this equation, generated by BCF, is
;; then formed and returned.

(define (make-input-equation equation checkpoints outputs)
  (bcf (edis (eliminate equation checkpoints) outputs)) )

;;
;; (COMBINE term inputs)
;;
;; Parameters:

```

```

;; term - A term from the f=1 form of the INPUT-EQUATION.
;; inputs - A list of the inputs of the circuit.
;;
;; -- All of the INPUTS may not exist as literals of TERM. COMBINE is
;; used to insert the INPUTS that are not literals of TERM into
;; term. Due to the nature of Boolean Algebra, these missing
;; literal can be arbitrarily set to 0 or 1. In this
;; implementation, the missing literals are set to 1.
;; -- For example, if the inputs were (A B C), and term
;; were A B = 1, then the equations A B C = 1 or A B C' = 1 would
;; both satisfy the constraints imposed by TERM. Thus, C can be
;; arbitrarily chosen. COMBINE sets C to 1.

(define (combine term inputs)
  (cond ( (null? inputs)
           '())
        ( (member (bar (car inputs)) term)
           (cons (bar (car inputs))
                 (combine term (cdr inputs)) ))
        ( else
           (cons (car inputs)
                 (combine term (cdr inputs)))))))

```

```

;;;;;;
;;          Filename: interp.s
;;
;; This module provides the facilities to interpret the output
;; equation from the TESTER.S module of the system. The facilities
;; provided include a procedure to compare the designed circuit to
;; the function that the circuit is actually performing, an
;; interpretation of the faults in the circuit, and a summary of
;; system metrics.
;;
;; NOTE: This implementation is based on the assumption of a single
;;       output circuit. Procedures must be revised to accomodate
;;       multiple output circuit diagnosis.
;;
;; Requires the files: boolean.fsl, eqn-gen.fsl, eqn-gena.fsl,
;;                      tokenize.fsl, interp-a.fsl
;;
;;;;;;
;; (INTERPRET intermediate-format phi tester-output)
;;
;; Parameters:
;;   intermediate-format - The data structure, a list in prefix-form
;;                         that was returned by procedure
;;                         RUN-INPUT-MODULE. This list is used to
;;                         determine the appropriate gate a given
;;                         fanout node is associated with when
;;                         printing out results for each node.
;;   phi - The data returned by GENERATE-EQUATION. The information
;;         provided by this list includes the circuit INPUTS,
;;         OUTPUTS, and CHECKPOINTS.
;;   tester-output - The data returned by TESTER. This includes the
;;                  NO-OF-TESTS that were conducted as well as the
;;                  FINAL-EQUATION generated by TESTER. This
;;                  equation is solved to yield the circuit FUNCTION
;;                  as well as the FAULT-CLASSES in the circuit.
;;
;; -- INTERPRET takes the output from all of the other major modules
;; and interprets the information to obtain the results of the
;; diagnostic test.
;; -- DISPLAY-FUNCTIONS is called to determine the function that the
;; circuit performed based on the diagnostic test, as opposed to
;; the function that it was designed to perform. An equivalency
;; check is made to compare the actual to the designed function.
;; -- INTERPRET-FAULTS is called to derive the faults in the circuit,
;; both those that can be positively determined as well as cases of
;; faults that may have occurred, but cannot be determined with
;; certainty.
;; -- DISPLAY-SYSTEM-METRICS is used to make a quick determination of
;; the a performance metrics of the diagnostic system.
;; -- Finally, the user is asked whether he would like to diagnose

```

```

;; another circuit. The REPLY, in the form of #T or '() is
;; returned by INTERPRET to the calling procedure where it is used
;; to determine whether to reexecute the calling module, or return
;; to the main menu of the diagnostic system.

(define (interpret intermediate-format phi tester-output)
  ; break down information from input parameters
  (let* ( (inputs      (cadr phi))
          (outputs     (cadadr phi))
          (checkpoints (cadaddr phi))
          (no-of-tests (car tester-output))
          (final-equation (cdr tester-output))
          (function    (solve-fcn final-equation checkpoints
                                    outputs))
          (fault-classes (solve-cps final-equation outputs)) )
    (newline)
    (writeln "***** Results *****")
    ; print out the function that the circuit is performing, the
    ; function that it is supposed to perform, and whether the
    ; two functions are equivalent.
    (display-functions function outputs intermediate-format)
    ; print out the possible faults in the circuit
    (interpret-faults checkpoints fault-classes
                        intermediate-format)
    ; display the performance metrics of the system
    (display-system-metrics inputs no-of-tests)
    (writeln "Would you like to try another circuit? ")
    (writeln "If so, type yes and <rtn>, else type no and <rtn>.")
    (writeln "A reply of no returns you to the main menu.")
    (display "Enter yes or no (default is no) --> ")
    (let ( (reply (read-line)) )
      (if (equal? reply "yes")
          #T
          '() ))))

;; (SOLVE-FCN equation checkpoints outputs)
;;
;; Parameters:
;;   equation - The final equation produced by procedure TESTER.
;;             This equation holds all information about the state
;;             of the system after it has been determined that no
;;             new information can be determined from further tests.
;;   checkpoints - A list of the checkpoint variables introduced into
;;                the equation.
;;   outputs - A list of the output nodes of the circuit.
;;
;; -- SOLVE-FCN is used to generate the equation that the circuit is
;;   performing based on the results of the input-output experiments.
;; -- The input EQUATION is of the form:

```

```

;;      EQUATION(x,y,z) = 0  where x is the input variables,
;;                          y is the checkpoint variables,
;;                          and z is a single output variable
;;
;; -- This EQUATION must then be converted to the form:
;;      a(x) z' + a'(x) z + g(y) = 0  where a(x) is a function of the
;;                                      input variables, and g(y) is a
;;                                      function of the checkpoint variables
;;
;; -- R(x) yields the actual circuit function. To obtain R(x), the
;; OUTPUTS can be DIVIDEd into the EQUATION using Boolean division.
;; This leaves an equation in terms of inputs and checkpoints.
;; Then the CHECKPOINTS can be removed using conjunctive
;; ELIMINATION to yield the single formula a(x).

(define (solve-fcn equation checkpoints outputs)
  (eliminate (divide equation outputs)
    checkpoints))

;;
;; (SOLVE-CPS equation outputs)
;;
;; Parameters:
;;   equation - The final equation produced by procedure TESTER.
;;             This equation holds all information about the state
;;             of the system after it has been determined that no
;;             new information can be determined from further
;;             input-output tests.
;;   outputs - A list of the output nodes of the circuit.
;;
;; -- SOLVE-CPS is used to generate the equation which can be solved
;; to determine the possible faults in the circuit. This equation
;; is based on the results of the input-output experiment.
;; -- EQUATION is of the form:
;;      EQUATION(x,y,z) = 0  where x is the input variables,
;;                          y is the checkpoint variables,
;;                          and z is a single output variable
;;
;; -- This EQUATION must then be converted to the form:
;;      a(x) z' + a'(x) z + g(y) = 0  where a(x) is a function of the
;;                                      input variables, and g(y) is a
;;                                      function of the checkpoint variables
;;
;; -- g(y) yields the possible faults function. To obtain g(y), the
;; OUTPUTS can be ELIMINATED from the EQUATION using conjunctive
;; elimination. This leaves an equation in terms of the
;; checkpoints.
;;
;; -- This equation is in f=0 form which is difficult to solve to
;; determine the states of the checkpoint variables. Thus, the
;; equation is COMPLEMENT to get the f=1 form. This equation is
;; SIMPLIFIED to yield an equation in which the terms represent the
;; possible faults in the circuit.
;;
;; -- Literals that exist in each of the terms are variables the state
;; of which has been positively determined. When these variables

```

```

;; are removed, the terms left represent the possible faults that
;; may exist in the circuit.

(define (solve-cps equation outputs)
  (simplify (complement (eliminate equation outputs))) )

;; (DISPLAY-FUNCTIONS function outputs intermediate-format)
;;
;; Parameters:
;;   function - The function that the circuit is performing as
;;             determined by SOLVE-FCN.
;;   outputs - A list of the outputs of the circuit.
;;   intermediate-format - The data structure, a list in prefix-form
;;                         that was returned by procedure
;;                         RUN-INPUT-MODULE. This list is used to
;;                         determine the function that the circuit
;;                         was designed to perform.
;;
;; -- DISPLAY-FUNCTIONS determines the circuit's ACTUAL-FUNCTION, the
;; circuit's DESIGNED-FUNCTION and prints these functions to the
;; screen in the form of a Boolean equation.
;; -- An equivalency test is made to determine if these functions are
;; equivalent. The result of this test is output to the screen.
;; -- The FUNCTION is XORed with the OUTPUTS to get ACTUAL-FUNCTION in
;; f=0 form. The prefix-form of the circuit, represented by the
;; INTERMEDIATE-FORMAT is used to determine the DESIGNED-FUNCTION.
;; The prefix-form must be reduced by MAKE-SOP and INTERNAL-NODES
;; must be ELIMINATED to yield an equation in the form of inputs
;; and outputs without internal nodes.
;; -- FUNCTION-D the DESIGNED-FUNCTION in the same form as the input
;; parameter FUNCTION to allow use of a single procedure,
;; DISPLAY-CIRCUIT-FUNCTION, in displaying the circuit function.
;; -- EQUIVALENCE-RESULT is the result of XORing the DESIGNED-FUNCTION
;; with the ACTUAL-FUNCTION. When two f=0 equation are XORed
;; together, if the result is 0, or in this representation '(),
;; ' then the equations are equivalent.

(define (display-functions function outputs intermediate-format)
  (let* ( (actual-function (bcf (xor function (list outputs))))
          (designed-function
            (eliminate (simplify (make-sop intermediate-format))
                      (get-internal-nodes intermediate-format)))
          (function-d (bcf (divide (mult designed-function
                                             (list (list outputs)))
                                     outputs)))
          (equivalence-result (xor designed-function
                                 actual-function)) )
    (newline)
    (writeln "The function that the circuit was designed to perform"

```

```

is: ")
    (newline)
    (display "    ")
    (display-circuit-function function-d outputs)
    (newline)
    (writeln "The function that the circuit is performing is: ")
    (newline)
    (display "    ")
    (display-circuit-function function outputs)
    (newline)
    (if (equal? equivalence-result '())
        (begin
            (display "The actual circuit IS equivalent to the ")
            (writeln "designed circuit."))
        (begin
            (display "The actual circuit IS NOT equivalent to the ")
            (writeln "designed circuit."))) )))

;; (DISPLAY-CIRCUIT-FUNCTION function outputs)
;;
;; Parameters:
;;   function - An equation representing the function of the circuit.
;;   outputs - The outputs of the circuit.
;;
;; -- DISPLAY-CIRCUIT-FUNCTION takes an equation representing the
;;   function that the circuit is performing, an displays this
;;   equation.
;; -- CONVERT-NODE-BACK is used to remove the suffix from the output
;;   node symbol so that it is output in the form of the original
;;   output symbol that was used by the user. This node is DISPLAYed
;;   followed by an equals sign.
;; -- DISPLAY-CIRCUIT-FUNCTION-1 is called to display the FUNCTION
;;   which is only in terms of the inputs.

(define (display-circuit-function function outputs)
  (let ( (output-node (convert-node-back (car outputs))) )
    (display output-node)
    (display " = ")
    (display-circuit-function-1 function)
    (newline) ))

;; (DISPLAY-CIRCUIT-FUNCTION-1 function)
;;
;; Parameters:
;;   function - A formula representing the function of the circuit.
;;
;; -- DISPLAY-CIRCUIT-FUNCTION-1 displays the circuit function.

```

```

;; -- FUNCTION is a list of the form:
;;   ((X1 (X2 X3) ((X1 X4) (X5) ((X6))))
;; which represents the formula:
;;   X1 X2'X3 + X1'X4 + X5 + X6'
;; Each of the top-level sublists is a term of this formula. If a
;; literal exists in the top-level sublist in the form of a
;; sublist, then it exists logically in complemented form;
;; uncomplemented otherwise.
;; -- FIRST-TERM is CARed from the FUNCTION and displayed by
;; DISPLAY-TERM.
;; -- If there are remaining terms in FUNCTION, then a + sign is
;; DISPLAYed, and DISPLAY-CIRCUIT-FUNCTION-1 is called recursively
;; to display the remaining terms of the formula.

(define (display-circuit-function-1 function)
  (if (null? function)
      (display-term function)
      (let ( (first-term (car function)) )
        (display-term (list first-term))
        (if (not (null? (cdr function)))
            (begin
              (display "+ ")
              (display-circuit-function-1 (cdr function)))
            '())))
    )

;; (DISPLAY-TERM term)
;;
;; Parameters:
;;   term - a list of the form (((X1) X2 (X3))) where each of the
;;         top level elements represents a term of a Boolean
;;         equation. The example list represents a single term
;;         X1'X2 X3'.
;;
;; -- DISPLAY-TERM prints a "1" if the term is of the form '() which
;; represents a Boolean 1.
;; -- DISPLAY-TERM prints a "0" if the term is of the form '() which
;; represents a Boolean 0.
;; -- If TERM is not of this form, DISPLAY-TERM-1 is called to display
;; TERM.

(define (display-term term)
  (cond ( (member nil term)
          (princ "1 " ) )
        ( (null? term)
          (princ "0 " ) )
        ( else
          (display-term-1 term) )))


```

```

;; (DISPLAY-TERM-1 term)
;;
;; Parameters:
;;   term - a list of the form (((X1) X2 (X3))) where each of the
;;          top level elements represents a term of a Boolean
;;          equation. The example list represents a single term
;;          X1'X2 X3'.
;;
;; -- If TERM is nil, then DISPLAY-TERM-1 returns '(). Otherwise, the
;;    TERM is sorted by SORT-TERM from file boolean.s. Then, the
;;    first term is displayed by DISPLAY-TERM-2. The remaining terms
;;    are displayed by a recursive call to DISPLAY-TERM-1.

(define (display-term-1 term)
  (cond ( (null? term)
           '() )
        ( else
           (display-term-2 (sort-term (car term)))
           (display-term-1 (cdr term)) )))

;;
;; (DISPLAY-TERM-2 term)
;;
;; Parameters:
;;   term - a list of the form ((X1) X2 (X3)) representing the term
;;          X1'X2 X3'.
;;
;; -- DISPLAY-TERM-2 takes a list representing a term and prints out
;;    each of the literals until the entire term has been output.
;; -- If a literal exists in the term as a sublist, then it is
;;    complemented, and a '"" (prime) is output immediately after the
;;    literal. Otherwise, a space is output after the literal.
;;    DISPLAY-TERM-2 is called recursively to output the remaining
;;    literals of the TERM.
;; -- CONVERT-NODE-BACK is called to remove the suffix from the nodes
;;    so that they are output in the form of the original node symbols
;;    used by the user.

(define (display-term-2 term)
  (cond ( (null? term)
           '() )
        ( (atom? (car term))
           (princ (convert-node-back (car term))) (princ " ")
           (display-term-2 (cdr term)) )
        ( else
           (princ (convert-node-back (car (car term)))) (princ "''")
           (display-term-2 (cdr term)) )))

```

```

;; (CONVERT-NODE-BACK node)
;;
;; Parameters:
;;   node - A symbol of the form ABC--.
;;
;; -- CONVERT-NODE-BACK accepts a NODE of the given form, removing the
;;   last two characters and returning a symbol of the form ABC.

(define (convert-node-back node)
  (let* ( (node-1 (string->list (symbol->string node)))
         (node-less-suffix (remove-suffix node-1)) )
    (string->symbol (list->string node-less-suffix)) ))


;; (INTERPRET-FAULTS checkpoints fault-classes intermediate-format)
;;
;; Parameters:
;;   checkpoints - A list of the checkpoint variables generated by
;;                 the system.
;;   fault-classes - A list of lists representing different fault
;;                  cases that may occur.
;;   intermediate-format - The data structure, a list in prefix-form
;;                         that was returned by procedure
;;                         RUN-INPUT-MODULE. This list is used to
;;                         determine the appropriate gate a given
;;                         fanout node is associated with when
;;                         printing out faults for each node.
;;
;; -- INTERPRET-FAULTS is called to derive the faults in the circuit,
;; both those that can be positively determined as well as cases of
;; faults that may have occurred, but cannot be determined with
;; certainty.
;;
;; -- REMOVE-LAST-CHAR-FROM-ALL-ELTS accepts the list of CHECKPOINTS
;; which is of the form (AX0 AX1 BO0 BO1 B10 B11 CX0 CX1) and
;; returns a list of the form (AX BO B1 CX). This latter list
;; represents the actual checkpoints in the circuit.
;;
;; GET-INPUT-CHECKPOINTS accepts the new list and returns a list of
;; the INPUT-CHECKPOINTS which is a list of the form (AX CX).
;;
;; GET-SUBLIST subtracts the INPUT-CHECKPOINTS list from the new
;; list to form the FANOUT-CHECKPOINTS list, which in this example
;; would be (BO B1). The fanout checkpoints must be distinguished
;; from the input checkpoints because the output of the faults for
;; these two distinct types of checkpoints is different. The input
;; nodes be only listed. The fanout node faults must have the gate
;; displayed also so the user knows which fanout stem may have a
;; fault.
;;
;; -- GET-NORMAL-NODES accepts the list of FAULT-CLASSES and
;; determines the normal nodes in the list. The second parameter
;; is the list of the nodes to check for normality. In the first

```

```

;; call to GET-NORMAL-NODES, the INPUT-CHECKPOINTS are checked to
;; see if they are normal. In the second call, the
;; FANOUT-CHECKPOINTS are checked.
;;
-- REMOVE-NORMAL-NODES is called to remove the NORMAL-INPUT-NODES
and the NORMAL-FANOUT-NODES from the fault classes, producing
FAULT-CLASSES-1 and FAULT-CLASSES-2, respectively.
;;
-- GET-COMMON-NODES gets all of the literals common to each of the
terms after the normal nodes have been removed.
REMOVE-COMMON-NODES removes the COMMON-NODES from
FAULT-CLASSES-2 to produce FAULT-CLASSES-3 which is a list of
terms which have no literals (sublists) in common. Each of
these terms represents a different fault that may have occurred
in the circuit.
;;
-- GET-COMMON-INPUT-NODES extracts the COMMON-INPUT-NODES from
the COMMON-NODES. GET-SUBLISTS subtracts the COMMON-INPUT-NODES
from the COMMON-NODES to get the COMMON-FANOUT-NODES. The
COMMON-INPUT-NODES and COMMON-FANOUT-NODES are used to get the
stuck-at-0, stuck-at-1, not-stuck-at-0, and not-stuck-at-1 nodes
for both the input and fanout nodes. Lists are made for each
case. In many cases, these list may be nil.
;;
-- SHOW-LIST-OF-NODES is called to print out the input nodes for
the appropriate fault. SHOW-FANOUTS is called to print out the
fanout nodes for the appropriate fault. SHOW-FANOUTS outputs
the appropriate node as well as the gate that the node is
associated with. A given fanout node may have a fanout of
three, each of which has an associated checkpoint. Thus, the
gate must be associated with the checkpoint when the checkpoint
fault status is output.
;;
-- Remaining fault cases, those that represent different faults
that may be occurring in the circuit are interpreted by a call
to INTERPRET-FAULT-CASES. CHECKPOINTS-1, the
INTERMEDIATE-FORMAT, and FAULT-CLASSES-3 are passed to
INTERPRET-FAULT-CASES.

```

```

(define (interpret-faults checkpoints fault-classes
                           intermediate-format)
  (let* ((checkpoints-1 (remove-last-char-from-all-elts checkpoints))
         (input-checkpoints (get-input-checkpoints checkpoints-1))
         (fanout-checkpoints (get-sublist checkpoints-1
                                           input-checkpoints))
         (normal-input-nodes (get-normal-nodes input-checkpoints
                                              fault-classes))
         (fault-classes-1 (remove-normal-nodes normal-input-nodes
                                              fault-classes))
         (normal-fanout-nodes (get-normal-nodes fanout-checkpoints
                                              fault-classes-1))
         (fault-classes-2 (remove-normal-nodes normal-fanout-nodes
                                              fault-classes-1))
         (prefix-list (make-unique-fanouts intermediate-format))
         (common-nodes (get-common-nodes fault-classes-2))
         (fault-classes-3 (remove-common-nodes common-nodes
                                              fault-classes-2)))
    (interpret-faults-1 prefix-list checkpoints-1
                        intermediate-format
                        fault-classes-3))

```

```

fault-classes-2))
(common-input-nodes (get-common-input-nodes common-nodes))
(common-fanout-nodes (get-sublist common-nodes
                                    common-input-nodes))
(input-nodes-s-a-0 (get-stuck-at-0-nodes
                                    common-input-nodes))
(input-nodes-s-a-1 (get-stuck-at-1-nodes
                                    common-input-nodes))
(input-nodes-n-s-a-0 (get-not-stuck-at-0-nodes
                                    common-input-nodes))
(input-nodes-n-s-a-1 (get-not-stuck-at-1-nodes
                                    common-input-nodes))
(fanout-nodes-s-a-0 (get-stuck-at-0-nodes
                                    common-fanout-nodes))
(fanout-nodes-s-a-1 (get-stuck-at-1-nodes
                                    common-fanout-nodes))
(fanout-nodes-n-s-a-0 (get-not-stuck-at-0-nodes
                                    common-fanout-nodes))
(fanout-nodes-n-s-a-1 (get-not-stuck-at-1-nodes
                                    common-fanout-nodes)) )

(newline)
(newline)
(display "**** The following information is certain ")
(writeln "about the circuit **** ")
(newline)
writeln "Input nodes (which do not fanout) that are normal:"
(newline)
(if (null? normal-input-nodes)
    (writeln "    --none--")
    (show-list-of-nodes normal-input-nodes))
(newline)
writeln "Input nodes (which do not fanout) that are stuck-at-
0:")
(newline)
(if (null? input-nodes-s-a-0)
    (writeln "    --none--")
    (show-list-of-nodes input-nodes-s-a-0))
(newline)
writeln "Input nodes (which do not fanout) that are stuck-at-
1:")
(newline)
(if (null? input-nodes-s-a-1)
    (writeln "    --none--")
    (show-list-of-nodes input-nodes-s-a-1))
(newline)
writeln "Input nodes (which do not fanout) that are NOT stuck-
at-0:")
(newline)
(if (null? input-nodes-n-s-a-0)
    (writeln "    --none--")
    (show-list-of-nodes input-nodes-n-s-a-0))

```

```

(newline)
(writeln "Input nodes (which do not fanout) that are NOT stuck-
at-1:")
(newline)
(if (null? input-nodes-n-s-a-1)
  (writeln "    --none--")
  (show-list-of-nodes input-nodes-n-s-a-1))
(newline)
(newline)
(writeln "Fanout nodes that are normal:")
(newline)
(if (null? normal-fanout-nodes)
  (writeln "    --none--")
  (show-fanouts normal-fanout-nodes prefix-list))
(newline)
(writeln "Fanout nodes that are stuck-at-0:")
(newline)
(if (null? fanout-nodes-s-a-0)
  (writeln "    --none--")
  (show-fanouts fanout-nodes-s-a-0 prefix-list))
(newline)
(writeln "Fanout nodes that are stuck-at-1:")
(newline)
(if (null? fanout-nodes-s-a-1)
  (writeln "    --none--")
  (show-fanouts fanout-nodes-s-a-1 prefix-list))
(newline)
(writeln "Fanout nodes that are NOT stuck-at-0:")
(newline)
(if (null? fanout-nodes-n-s-a-0)
  (writeln "    --none--")
  (show-fanouts fanout-nodes-n-s-a-0 prefix-list))
(newline)
(writeln "Fanout nodes that are NOT stuck-at-1:")
(newline)
(if (null? fanout-nodes-n-s-a-1)
  (writeln "    --none--")
  (show-fanouts fanout-nodes-n-s-a-1 prefix-list))
; interpret the remaining cases, if they exist
(newline)
(if (not (equal? fault-classes-3 '()))
  (interpret-fault-cases checkpoints-1
    fault-classes-3
    intermediate-format)))

```

```

;; (SHOW-LIST-OF-NODES nodes)
;;
;; Parameters:
;;   nodes - a list of nodes
;;
;; -- SHOW-LIST-OF-NODES accepts a list of the form (AX BX CX) and
;;   removes the last character from each of the symbols to produce a
;;   MODIFIED-LIST of the form (A B C).
;; -- SHOW-NODES is then called to display each of the nodes in the
;;   new list.

(define (show-list-of-nodes nodes)
  (let* ( (modified-list (remove-last-char-from-all-elts nodes)) )
    (show-nodes modified-list)))

;;
;; (SHOW-NODES lst)
;;
;; Parameters:
;;   lst - an arbitrary list
;;
;; -- SHOW-NODES displays each of the elements of LST on a separate
;;   line, until there are no further elements to display.

(define (show-nodes lst)
  (if (null? lst)
      '()
      (begin
        (display "    ")
        (writeln (car lst))
        (show-nodes (cdr lst)))))

;;
;; (REMOVE-LAST-CHAR-FROM-ALL-ELTS lst)
;;
;; Parameters:
;;   lst - an arbitrary list
;;
;; -- REMOVE-LAST-CHAR-FROM-ALL-ELTS removes that last character from
;;   every symbol in an arbitrary list.  The procedure breaks down
;;   sublists to change every symbol at any level.
;; -- It is assumed that every symbol has two or more characters.

(define (remove-last-char-from-all-elts lst)
  (cond ( (null? lst)
           '())
        ( (symbol? lst)

```

```

(remove-last-char-from-symbol lst) )
( (symbol? (car lst))
  (remove-duplicates
    (cons (remove-last-char-from-symbol (car lst))
          (remove-last-char-from-all-elts (cdr lst)))) )
( else
  (remove-duplicates
    (cons (remove-last-char-from-all-elts (car lst))
          (remove-last-char-from-all-elts (cdr lst)))) )))

;; (SHOW-FANOUTS fanout-nodes prefix-list)
;;
;; Parameters:
;;   fanout-nodes - a list of nodes of the form (B1 B0)
;;   prefix-list - The prefix list of the input circuit; this list
;;                 was modified to MAKE-UNIQUE-FANOUTS of each of the
;;                 fanout nodes. This is necessary to distinguish
;;                 the fanouts and associate them with the list of
;;                 fanout nodes.
;;
;; -- SHOW-FANOUT takes the PREFIX-LIST and removes that last char
;; from each of the NODE-SYMBOLS. This leaves a list of the form:
;;
;;   ((EQ E- (NOT (* AX BO)))
;;    (EQ F- (NOT B1))
;;    (EQ Z- (NOT (* E- F-))))
;;
;; -- SHOW-FANOUTS-1 is then passed the list of FANOUT-NODES and the
;; new prefix list.

(define (show-fanouts fanout-nodes prefix-list)
  (let ( (prefix-list-1 (remove-last-char-from-node-symbols
                                         prefix-list)) )
    (show-fanouts-1 fanout-nodes prefix-list-1) )

;; (SHOW-FANOUTS-1 fanout-nodes prefix-list)
;;
;; Parameters:
;;   fanout-nodes - A list of fanout nodes
;;   prefix-list - the modified prefix-list from SHOW-FANOUTS
;;
;; -- SHOW-FANOUTS-1 iteratively outputs each of the FANOUT-NODES in
;; the input list, displaying in sequence the NODE and then the
;; GATE associated with that fanout node.
;; -- GET-GATE returns the EQUATION associated with the fanout node.
;; For example, if node B0 were to be displayed, then the EQUATION
;; returned by GET-GATE would be (EQ E- (NOT (* AX BO))). This

```

```

;;      equation is displayed by SHOW-EQUATION.
;; -- SHOW-FANOUTS-1 calls itself recursively until all of the
;;   FANOUT-NODES in the original list have been displayed with the
;;   appropriate gate.

(define (show-fanouts-1 fanout-nodes prefix-list)
  (if (null? fanout-nodes)
      '()
      (let ( (equation (get-gate (car fanout-nodes) prefix-list)) )
        (display "    Node ")
        (display (remove-last-char-from-symbol (car fanout-nodes)))
        (display " of gate: ")
        (show-equation equation)
        (show-fanouts-1 (cdr fanout-nodes) prefix-list) )))

;; (SHOW-EQUATION equation)
;;
;; Parameters:
;;   equation - a list of the form (EQ E- (NOT (* AX BO)))
;;
;; -- SHOW-EQUATION displays the above equation in the form E = A * B.
;; -- First, the output node for the gate is display followed by an
;;   equals sign. Then either SHOW-NEGATED-EQUATION, or SHOW-FORMULA
;;   are called depending on whether the gate is of the NEGATED
;;   variety, i.e. NAND.

(define (show-equation equation)
  (let* ( (output (cadr equation))
          (input (caddr equation)) )
    (display (remove-last-char-from-symbol output)))
  (display " = ")
  (if (equal? (car input) 'NOT)
      (show-negated-equation (cadr input))
      (show-formula input))
  (newline) ))

;;
;; (INTERPRET-FAULT-CASES checkpoints fault-classes
;;                                intermediate-format)
;;
;; Parameter:
;;   checkpoints - A list of the form (AX BO BI CX).
;;   fault-classes - A list of lists in which each sublist is a term
;;                  representing a distinct fault class.
;;   intermediate-format - the intermediate-format from
;;                        RUN-INPUT-MODULE.
;;
;; -- INTERPRET-FAULT-CASES prints out an introductory message and

```

```

;; then calls INTERPRET-FAULT-CASES-1. All parameters are passed.
;; A new parameter, the number 1 is passed to INTERPRET-FAULT-CASES
;; which uses this number to keep track of the different fault
;; cases.

(define (interpret-fault-cases checkpoints fault-classes
                                intermediate-format)

  (newline)
  (writeln "**** One of the following cases holds for the circuit
****")
  (newline)
  (interpret-fault-cases-1 checkpoints fault-classes
                            intermediate-format 1) )

;; (INTERPRET-FAULT-CASES-1 checkpoints fault-classes
;;                               intermediate-format
;;                               case-number)

;;
;; Parameters:
;;   checkpoints - A list of the form (AX BO B1 CX).
;;   fault-classes - A list of lists in which each sublist is a term
;;                   representing a distinct fault class.
;;   intermediate-format - The intermediate-format from
;;                         RUN-INPUT-MODULE.
;;   case-number - An integer. Initially, this number is 1. Every
;;                 time that INTERPRET-FAULT-CASES-1 is called
;;                 recursively to interpret another case, this number
;;                 is incremented.

;;
-- INTERPRET-FAULT-CASES-1 operates similarly to INTERPRET-FAULTS.
-- However, it is tailored to interpreting distinct fault cases
-- that may have occurred in the circuit.
-- The first term (FIRST-CASE) is removed from the list of
-- FAULT-CLASSES. FIRST-CASE is examined to determine the types of
-- faults associated with each of the nodes in the term. The order
-- this is done is the same as in INTERPRET-FAULTS. There is no
-- need to check for COMMON-NODES, because no common nodes exist
-- between terms of FAULT-CLASSES when this procedure is invoked.
-- The only case found different in this procedure than in
-- INTERPRET-FAULTS is that nodes may be found that have been
-- interpreted to be stuck-at-0 or stuck-at-1 in which the
-- complementary not-stuck-at-1 or not-stuck-at-0, respectively,
-- variable is not found. In this case, lists are made of "only"
-- stuck-at-0 or "only" stuck-at-1 nodes. However, the display
-- procedures do not differentiate between stuck-at-0 and
-- only-stuck-at-0 and stuck-at-1 and only-stuck-at-1.
-- For each case of faults, the input node and fanout node faults
-- are displayed together. SHOW-INPUT-NODES is called to display

```

```

;;      input nodes, and SHOW-FANOUT-NODES is called to display the
;;      fanout nodes and associated gates.
;; -- After a case is interpreted and displayed, then
;;   INTERPRET-FAULT-CASES-1 calls itself recursively until all cases
;;   have been displayed. CASE-NUMBER is incremented with each
;;   recursive call.

(define (interpret-fault-cases-1 checkpoints
                                    fault-classes
                                    intermediate-format
                                    case-number)

  (if (not (null? fault-classes))
      (let* ((first-case (list (car fault-classes)))
             (input-checkpoints (get-input-checkpoints checkpoints))
             (fanout-checkpoints (get-sublist checkpoints
                                              input-checkpoints))
             (normal-input-nodes (get-normal-nodes input-checkpoints
                                                    first-case))
             (first-case-1 (remove-normal-nodes normal-input-nodes
                                                first-case))
             (normal-fanout-nodes (get-normal-nodes fanout-checkpoints
                                                    first-case-1))
             (first-case-2 (remove-normal-nodes normal-fanout-nodes
                                                first-case-1))
             (prefix-list (make-unique-fanouts intermediate-format))
             (prefix-list-1 (remove-last-char-from-node-symbols
                            prefix-list))
             (common-nodes (get-common-nodes first-case-2))
             (input-faults (get-common-input-nodes common-nodes))
             (fanout-faults (get-sublist common-nodes input-faults))
             (input-nodes-s-a-0 (get-stuck-at-0-nodes input-faults))
             (input-nodes-s-a-1 (get-stuck-at-1-nodes input-faults))
             (input-nodes-n-s-a-0 (get-not-stuck-at-0-nodes
                                   input-faults))
             (input-nodes-n-s-a-1 (get-not-stuck-at-1-nodes
                                   input-faults))
             (input-nodes-o-s-a-0 (get-only-stuck-at-0-nodes
                                   input-faults))
             (input-nodes-o-s-a-1 (get-only-stuck-at-1-nodes
                                   input-faults))
             (fanout-nodes-s-a-0 (get-stuck-at-0-nodes fanout-faults))
             (fanout-nodes-s-a-1 (get-stuck-at-1-nodes fanout-faults))
             (fanout-nodes-n-s-a-0 (get-not-stuck-at-0-nodes
                                   fanout-faults))
             (fanout-nodes-n-s-a-1 (get-not-stuck-at-1-nodes
                                   fanout-faults))
             (fanout-nodes-o-s-a-0 (get-only-stuck-at-0-nodes
                                   fanout-faults))
             (fanout-nodes-o-s-a-1 (get-only-stuck-at-1-nodes
                                   fanout-faults))))

```

```

(writeln "      **** Case #" case-number " ****")
(newline)
(if (not (null? normal-input-nodes))
    (show-input-nodes normal-input-nodes 'normal))
(if (not (null? input-nodes-s-a-0))
    (show-input-nodes input-nodes-s-a-0 'stuck-at-0))
(if (not (null? input-nodes-s-a-1))
    (show-input-nodes input-nodes-s-a-1 'stuck-at-1))
(if (not (null? input-nodes-n-s-a-0))
    (show-input-nodes input-nodes-n-s-a-0
                      'not-stuck-at-0))
(if (not (null? input-nodes-n-s-a-1))
    (show-input-nodes input-nodes-n-s-a-1
                      'not-stuck-at-1))
(if (not (null? input-nodes-o-s-a-0))
    (show-input-nodes input-nodes-o-s-a-0 'only-stuck-at-0))
(if (not (null? input-nodes-o-s-a-1))
    (show-input-nodes input-nodes-o-s-a-1 'only-stuck-at-1))
(if (not (null? normal-fanout-nodes))
    (show-fanout-nodes normal-fanout-nodes
                       'normal
                       prefix-list-1))
(if (not (null? fanout-nodes-s-a-0))
    (show-fanout-nodes fanout-nodes-s-a-0
                      'stuck-at-0
                      prefix-list-1))
(if (not (null? fanout-nodes-s-a-1))
    (show-fanout-nodes fanout-nodes-s-a-1
                      'stuck-at-1
                      prefix-list-1))
(if (not (null? fanout-nodes-n-s-a-0))
    (show-fanout-nodes fanout-nodes-n-s-a-0
                      'not-stuck-at-0
                      prefix-list-1))
(if (not (null? fanout-nodes-n-s-a-1))
    (show-fanout-nodes fanout-nodes-n-s-a-1
                      'not-stuck-at-1
                      prefix-list-1))
(if (not (null? fanout-nodes-o-s-a-0))
    (show-fanout-nodes fanout-nodes-o-s-a-0
                      'only-stuck-at-0
                      prefix-list-1))
(if (not (null? fanout-nodes-o-s-a-1))
    (show-fanout-nodes fanout-nodes-o-s-a-1
                      'only-stuck-at-1
                      prefix-list-1))

(newline)
(display "Press <return> to continue.")
(pause)
(newline)
(newline)

```

```

(interpret-fault-cases-1 checkpoints
  (cdr fault-classes)
  intermediate-format
  (1+ case-number)) )))

;; (REMOVE-LAST-CHAR-FROM-SYMBOL symbol)
;;
;; Parameters:
;;   symbol - an arbitrary symbol
;;
;; -- REMOVE-LAST-CHAR-FROM-SYMBOL decomposes the symbol, drops the
;;   last character from the symbol, reassembles the symbol and
;;   returns the NEW-SYMBOL.

(define (remove-last-char-from-symbol symbol)
  (let* ( (symbol-1 (string->list (symbol->string symbol)))
          (new-list (drop-last-char symbol-1))
          (new-symbol (string->symbol (list->string new-list))) )
    new-symbol))

;; (GET-INPUT-CHECKPOINTS checkpoints)
;;
;; Parameters:
;;   checkpoints - A list of the form (AX BO B1 CX).
;;
;; -- GET-INPUT-CHECKPOINTS returns a list in which the last character
;;   of every symbol is an X. The distinguishes primary input
;;   checkpoints from other checkpoints in the system.
;;   LAST-CHAR-EQ-X? is used to determine whether a given symbol has
;;   a last character of X.
;; -- For the given list, (AX CX) would be returned.

(define (get-input-checkpoints checkpoints)
  (cond ( (null? checkpoints)
           '())
        ( else
          (if (last-char-eq-x? (car checkpoints))
              (cons (car checkpoints)
                    (get-input-checkpoints (cdr checkpoints)))
              (get-input-checkpoints (cdr checkpoints)))))))

;; (LAST-CHAR-EQ-X? symbol)
;;
;; Parameter:
;;   symbol - an arbitrary symbol

```

```

;;
;; -- LAST-CHAR-EQ-X? decomposes the symbol.  GET-LAST-ELT is used to
;; get the last element from the list of characters (SYMBOL-L) that
;; comprise the SYMBOL.  If the LAST-CHAR equals X, then #T is
;; returned.  Otherwise, '() is returned.

(define (last-char-eq-x? symbol)
  (let* ( (symbol-l (string->list (symbol->string symbol)))
          (last-char (get-last-elt symbol-l)) )
    (equal? last-char '(\#X)) )

;;
;; (GET-NORMAL-NODES checkpoints fault-classes)
;;
;; Parameters:
;; checkpoints - A list of checkpoints, either of the form (AX BX)
;;               or (CO C1).
;; fault-classes - A list of lists in which each top level sublist
;;                  represents a set of faults that may have
;;                  occurred in the circuit.
;;
;; -- GET-NORMAL-NODES takes a node from the lists of checkpoints and
;; tests to see whether it is normal by calling NORMAL-NODE?.
;; -- If the given checkpoint is normal, it is added to the list that
;; is returned.  Otherwise, it is not.

(define (get-normal-nodes checkpoints fault-classes)
  (cond ( (null? checkpoints)
           '())
        ( (normal-node? (car checkpoints) fault-classes)
           (cons (car checkpoints)
                 (get-normal-nodes (cdr checkpoints) fault-classes)) )
        ( else
           (get-normal-nodes (cdr checkpoints) fault-classes) )))

;;
;; (NORMAL-NODE? checkpoint fault-classes)
;;
;; Parameters:
;; checkpoint - a single checkpoint symbol
;; fault-classes - A list of lists in which each top level sublist
;;                  represents a set of faults that may have
;;                  occurred in the circuit.
;;
;; -- NORMAL-NODE? takes a checkpoint symbol of the form AX or BO and
;; creates the symbols AX0 and AX1, or BO0 and BO1, respectively.
;; -- Then MEMBER-ALL-LISTS? is called to see if the complemented form
;;   of each of these variables is in every one of the sublists.
;; -- If both complemented forms are in every sublist, then and only

```

```

;; then is that node normal.

(define (normal-node? checkpoint fault-classes)
  (let* ( (checkpoint-1 (string->list (symbol->string checkpoint)))
          (checkpoint-0 (append checkpoint-1
                                 (string->list
                                   (number->string 0 '(int))))))
    (checkpoint-1 (append checkpoint-1
                           (string->list
                             (number->string 1 '(int)))))
      (symbol-0 (string->symbol (list->string checkpoint-0)))
      (symbol-1 (string->symbol (list->string checkpoint-1))) )
  (and (member-all-lists? (list symbol-0) fault-classes)
       (member-all-lists? (list symbol-1) fault-classes)) )

;; (REMOVE-NORMAL-NODES normal-nodes fault-classes)
;;
;; Parameters:
;;   normal-nodes - A list produced by GET-NORMAL-NODES of the form
;;                 (AX BX) or (CO C1).
;;   fault-classes - A list of lists in which each top level sublist
;;                  represents a set of faults that may have
;;                  occurred in the circuit.
;;
;; -- REMOVE-NORMAL-NODES removes the NORMAL-NODES from the
;; FAULT-CLASSES to produce a new list of fault classes with all of
;; the NORMAL-NODES removed.
;; -- REMOVE-NORMAL-NODES-1 is called to modify the list of
;; FAULT-CLASSES for a single node. REMOVE-NORMAL-NODES calls
;; itself recursively until all NORMAL-NODES have been removed from
;; the list of FAULT-CLASSES.

(define (remove-normal-nodes normal-nodes fault-classes)
  (if (null? normal-nodes)
      fault-classes
      (remove-normal-nodes (cdr normal-nodes)
                           (remove-normal-nodes-1 (car normal-nodes)
                                                 fault-classes)))))

;; (REMOVE-NORMAL-NODES-1 node fault-classes)
;;
;; Parameters:
;;   node - A node of the form AX or BO.
;;   fault-classes - The list of lists representing fault classes.
;;
;; -- REMOVE-NORMAL-NODES-1 takes the input NODE and creates the
;; appropriate checkpoint symbols, for AX this would be AX0 and

```

```

;; AX1, and uses a Boolean DIVIDE to remove these symbols from
;; every term.

(define (remove-normal-nodes-1 node fault-classes)
  (let* ( (node-1  (string->list (symbol->string node)))
          (node-0  (append node-1
                            (string->list
                                (number->string 0 '(int))))))
    (node-1  (append node-1
                      (string->list
                        (number->string 1 '(int))))))
    (symbol-0 (list (string->symbol (list->string node-0))))
    (symbol-1 (list (string->symbol (list->string node-1)))) )
  (divide (divide fault-classes symbol-0) symbol-1) )

;; (GET-COMMON-NODES lst)
;;
;; Parameters:
;; lst - A list of lists representing different fault classes.
;;
;; -- GET-COMMON-NODES returns a list of those items common to all
;; of the top-level sublists. If there is only one top-level
;; sublist, then it is returned. If there are more than one, then
;; GET-COMMON-NODES-1 is called and passed both the FIRST-LST as
;; well as the REST of the sublists.

(define (get-common-nodes lst)
  (let ( (first-lst (car lst))
         (rest      (cdr lst)) )
    (if (null? rest)
        first-lst
        (get-common-nodes-1 first-lst rest) )))

;; (GET-COMMON-NODES-1 first-lst list-of-lists)
;;
;; Parameters:
;; first-lst - One of the fault cases.
;; list-of-lists - All of the remaining fault cases.
;;
;; -- GET-COMMON-NODES-1 works by taking each element of the FIRST-LST
;; and checks to see if an element is the MEMBER-ALL-LISTS? of each
;; of the other lists of faults. If it is, then that element is
;; common to all of the fault cases.
;; -- If an element is not a MEMBER-ALL-LISTS?, then it is not common
;; to all of the fault cases. Only those elements that are common
;; to all of the fault cases are returned.
;; -- GET-COMMON-NODES-1 calls itself recursively until all of the

```

```

;;      elements of FIRST-LST in the initial call to GET-COMMON-NODES-1
;;      have been checked with respect to the other lists.

(define (get-common-nodes-1 first-lst list-of-lists)
  (if (null? first-lst)
      '()
      (let* ( (first-elt (car first-lst))
              (rest      (cdr first-lst)) )
            (if (member-all-lists? first-elt list-of-lists)
                (cons first-elt
                      (get-common-nodes-1 rest list-of-lists))
                (get-common-nodes-1 rest list-of-lists)))))

;; (MEMBER-ALL-LISTS? elt list-of-lists)
;;
;; Parameters:
;;   elt - an arbitrary element
;;   list-of-lists - an arbitrary list of lists
;;
;; -- MEMBER-ALL-LISTS? works by determining whether the element is a
;; member of the first list in the LIST-OF-LISTS. If it is, then
;; MEMBER-ALL-LISTS? calls itself recursively. If it calls itself
;; until LIST-OF-LISTS is exhausted, then ELT had to be a member of
;; all of the sublists in LIST-OF-LISTS.

(define (member-all-lists? elt list-of-lists)
  (if (null? list-of-lists)
      #t
      (if (member elt (car list-of-lists))
          (member-all-lists? elt (cdr list-of-lists))
          '())))

;; (REMOVE-COMMON-NODES lst list-of-lists)
;;
;; Parameters:
;;   lst - A list of elements common to each sublist of LIST-OF-LISTS
;;         that are to be removed from LIST-OF-LISTS.
;;   list-of-lists - An arbitrary list of lists.
;;
;; -- REMOVE-COMMON-NODES removes all of the elements of LST from each
;; of the top-level sublists of LIST-OF-LISTS. The first element
;; of LST is DIVIDEd into LIST-OF-LISTS to form a new list of
;; lists.
;; -- This new list, with the remaining elements of LST are then
;; passed to a recursive call of REMOVE-COMMON-NODES. This
;; continues until the elements of LST have been exhausted.

```

```

(define (remove-common-nodes lst list-of-lists)
  (if (null? lst)
      list-of-lists
      (remove-common-nodes (cdr lst) (divide list-of-lists
                                                (car lst)))))

;; (REMOVE-LAST-CHAR-FROM-NODE-SYMBOLS lst)
;;
;; Parameters:
;;   lst - a list of the form:
;;
;;     ((EQ E-- (NOT (* AX- BO-)))
;;      (EQ F-- (NOT B1-))
;;      (EQ Z-- (NOT (* E-- F--))))
;;
;; -- REMOVE-LAST-CHAR-FROM-NODE-SYMBOLS takes the LST and removes the
;; last character from each of the node symbols. This leaves a
;; list of the form:
;;
;;     ((EQ E- (NOT (* AX BO)))
;;      (EQ F- (NOT B1))
;;      (EQ Z- (NOT (* E- F-))))
;;
;; -- The LST is decomposed recursively until a symbol is reached.
;; Then if a node symbol is detected, the last character is
;; removed. Otherwise, the symbol is unchanged. The returned
;; list is the original list reassembled with the last character
;; removed from each of the node symbols.

(define (remove-last-char-from-node-symbols lst)
  (cond ( (null? lst)
           '())
        ( (symbol? lst)
           (if (good-symbol? lst)
               (remove-last-char-from-symbol lst)
               lst) )
        ( (symbol? (car lst))
           (if (good-symbol? (car lst))
               (cons (remove-last-char-from-symbol (car lst))
                     (remove-last-char-from-node-symbols (cdr lst)))
               (cons (car lst)
                     (remove-last-char-from-node-symbols (cdr lst))))))
        ( else
           (cons (remove-last-char-from-node-symbols (car lst))
                 (remove-last-char-from-node-symbols (cdr lst)))))))

```

```

;;;;;;
;;          Filename: interp-a.s           ;;
;;
;; This module is a continuation of file interp.s.  The procedures      ;;
;; provided in this file are the remaining procedures needed for      ;;
;; interpreting the faults in the circuit, as well as the summary of   ;;
;; system metrics procedures.                                         ;;
;;
;; NOTE: This implementation is based on the assumption of a single    ;;
;;       output circuit. Procedures must be revised to accomodate      ;;
;;       multiple output circuit diagnosis.                           ;;
;;
;; Requires the files: boolean.fsl, eqn-gen.fsl, eqn-gena.fsl,        ;;
;;                      tokenize.fsl, interp.fsl                   ;;
;;
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (GET-COMMON-INPUT-NODES lst)
;;
;; Parameters:
;;   lst - A list of the form ((AX0) AX1 (BOO) B11)
;;
;; -- GET-COMMON-INPUT-NODES accepts a LST of the given form and
;;   returns a list of those nodes that are input nodes. In this
;;   case, this would be the list ((AX0) AX1).
;; -- LST is decomposed until all symbols are tested to see whether
;;   their NEXT-TO-LAST-CHAR-EQ-Z?.
;; -- GET-COMMON-INPUT-NODES calls itself recursively until all nodes
;;   have been checked.
;;
(define (get-common-input-nodes lst)
  (cond ( (null? lst)
           '())
        ( (symbol? lst)
           (if (next-to-last-char-eq-x? lst)
               lst
               '()))
        ( (symbol? (car lst))
           (if (next-to-last-char-eq-x? (car lst))
               (cons (car lst) (get-common-input-nodes (cdr lst)))
               (get-common-input-nodes (cdr lst)) ) )
        ( else
           (if (next-to-last-char-eq-x? (caar lst))
               (cons (get-common-input-nodes (car lst))
                     (get-common-input-nodes (cdr lst)))
               (get-common-input-nodes (cdr lst)))) )
;;
;; (NEXT-TO-LAST-CHAR-EQ-X? symbol)
;;

```

```

;; Parameter:
;;   symbol - an arbitrary symbol
;;
;; -- NEXT-TO-LAST-CHAR-EQ-X? decomposes the SYMBOL and calls
;;   GET-NEXT-TO-LAST-ELT to get the next to last character of the
;;   list of characters that make up the SYMBOL.
;; -- If this character is X, then #T is returned; otherwise '().
(define (next-to-last-char-eq-x? symbol)
  (let* ( (symbol-1 (string->list (symbol->string symbol)))
          (next-to-last-char (get-next-to-last-elt symbol-1)) )
    (equal? next-to-last-char '#\X) ))


;; (GET-STUCK-AT-0-NODES list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form
;;                 ((AX1) AX0 B00 (B01)).
;;
;; -- GET-STUCK-AT-0-NODES gets the initial LIST-OF-NODES and calls
;;   GET-STUCK-AT-0-NODES-1 passing it two copies of the
;;   LIST-OF-NODES.
;; -- This was done to make the call less awkward from the
;;   INTERPRETATION modules.

(define (get-stuck-at-0-nodes list-of-nodes)
  (get-stuck-at-0-nodes-1 list-of-nodes list-of-nodes) )




;; (GET-STUCK-AT-0-NODES-1 list-of-nodes initial-list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form
;;                 ((AX1) AX0 B00 (B01)).
;;   initial-list-of-nodes - a list of the same form as
;;                         list-of-nodes.
;;
;; -- GET-STUCK-AT-0-NODES-1 scans the LIST-OF-NODES to see if a symbol
;; is found that is not in a sublist. Stuck-at-faults are those
;; variables not enclosed in a sublist.
;; -- Once a symbol is found at the top-level of LIST-OF-NODES, it is
;; checked to determine if the LAST-CHAR-EQ-0?. If so, then this
;; signifies a stuck-at-0 fault. Finally, to positively assert
;; that the checkpoint is stuck-at-0, the complementary
;; not-stuck-at-1 variable must be found by
;; EXISTS-NOT-STUCK-AT-PAIR?. In the given example, there exist two
;; stuck-at-0 cases, AX and B0. This is because AX0 and B00 exist
;; as symbols while their complementary stuck-at-1 pairs are in

```

```

;; sublists, which means that the checkpoints are not stuck-at-1.
;; The returned list in this case would be (AX BO). Once the two
;; different stuck-at variables have been used to determine that a
;; node is stuck-at-0, then the two distinct variables are no
;; longer necessary. REMOVE-LAST-CHAR-FROM-SYMBOL removes the
;; extraneous character.
;; -- The INITIAL-LIST-OF-NODES is scanned to find the stuck-at-0
;; variable in symbol form. The INITIAL-LIST-OF-NODES is carried
;; along because when checking for the complementary stuck-at-1
;; variable, it may occur either before or after the stuck-at-0
;; variable in the list.

(define (get-stuck-at-0-nodes-1 list-of-nodes initial-list-of-nodes)
  (cond ( (null? list-of-nodes)
           '() )
        ( (symbol? (car list-of-nodes))
            (if (and (last-char-eq-0? (car list-of-nodes))
                      (exists-not-stuck-at-pair? (car list-of-nodes)
                                                  initial-list-of-nodes))
                  (cons (remove-last-char-from-symbol
                           (car list-of-nodes))
                        (get-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                               initial-list-of-nodes)))
                (get-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                       initial-list-of-nodes)) )
        ( else
          (get-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                 initial-list-of-nodes) )))

;; (GET-STUCK-AT-1-NODES list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form
;;                 ((AX0) AX1 BO1 (BO0)).
;;
;; -- GET-STUCK-AT-1-NODES gets the initial LIST-OF-NODES and calls
;;   GET-STUCK-AT-1-NODES-1 passing it two copies of the
;;   LIST-OF-NODES.
;; -- This was done to make the call less awkward from the
;;   INTERPRETATION modules.

(define (get-stuck-at-1-nodes list-of-nodes)
  (get-stuck-at-1-nodes-1 list-of-nodes list-of-nodes) )

;;
;; (GET-STUCK-AT-1-NODES-1)
;;
;; Parameters:

```

```

;;      list-of-nodes - A list of nodes of the form
;;                      ((AX0) AX1 BO1 (BO0)).
;;      initial-list-of-nodes - a list of the same form as
;;                            list-of-nodes.

;;
;; -- GET-STUCK-AT-1-NODES-1 scans the LIST-OF-NODES to see a symbol
;; is found that is not in a sublist. Stuck-at-faults are those
;; variables not enclosed in a sublist.

;;
;; -- Once a symbol is found at the top-level of LIST-OF-NODES, it is
;; checked to determine if the LAST-CHAR-EQ-0? is not true. If so,
;; then this signifies a stuck-at-1 fault. Finally, to positively
;; assert that the checkpoint is stuck-at-1, the complementary
;; not-stuck-at-0 variable must be found by
;; EXISTS-NOT-STUCK-AT-PAIR? In the given example, there exist two
;; stuck-at-1 cases, AX and BO. This is because AX0 and BO0 exist
;; as symbols while their complementary stuck-at-0 pairs are in
;; sublists, which means that the checkpoints are not stuck-at-0.
;; The returned list in this case would be (AX BO). Once the two
;; different stuck-at variables have been used to determine that a
;; node is stuck-at-1, then the two distinct variables are no
;; longer necessary. REMOVE-LAST-CHAR-FROM-SYMBOL removes the
;; extraneous character.

;;
;; -- The INITIAL-LIST-OF-NODES is scanned to find the stuck-at-1
;; variable in symbol form. The INITIAL-LIST-OF-NODES is carried
;; along because when checking for the complementary stuck-at-0
;; variable, it may occur either before or after the stuck-at-1
;; variable in the list.

(define (get-stuck-at-1-nodes-1 list-of-nodes initial-list-of-nodes)
  (cond ( (null? list-of-nodes)
           '() )
        ( (symbol? (car list-of-nodes))
            (if (and (not (last-char-eq-0? (car list-of-nodes)))
                      (exists-not-stuck-at-pair? (car list-of-nodes)
                                                  initial-list-of-nodes))
                  (cons (remove-last-char-from-symbol
                           (car list-of-nodes))
                        (get-stuck-at-1-nodes-1 (cdr list-of-nodes)
                                               initial-list-of-nodes)))
                (get-stuck-at-1-nodes-1 (cdr list-of-nodes)
                                       initial-list-of-nodes)) )
        ( else
          (get-stuck-at-1-nodes-1 (cdr list-of-nodes)
                                 initial-list-of-nodes) )))

;;
;; (EXISTS-NOT-STUCK-AT-PAIR? node list-of-nodes)
;;
;; Parameters:
;;   node - a symbol of either the form AX0 or BX1

```

```

;;      list-of-nodes - a list of the form (AX0 (AX1) BO1 (BO0)).
;;
;; -- EXISTS-NOT-STUCK-AT-PAIR? takes the input NODE and creates the
;; opposite symbol regardless of the input form. Then depending on
;; the last character of the NODE, the appropriate symbol is used
;; to see whether it is in the LIST-OF-NODES.
;; -- For example, if AX0 were the input NODE, then a test would be
;; made to determine whether (AX1) is a member of LIST-OF-NODES.
;; -- IF BO1 were the NODE, then a test would be made to determine
;; whether (BO0) was a member of LIST-OF-NODES.

(define (exists-not-stuck-at-pair? node list-of-nodes)
  (let* ( (node-less-last-char (remove-last-char-from-symbol node))
          (node-1 (string->list
                        (symbol->string node-less-last-char)))
          (node-0 (append node-1
                        (string->list (number->string 0 '(int))))))
    (node-1 (append node-1
                    (string->list (number->string 1 '(int))))))
    (symbol-0 (string->symbol (list->string node-0)))
    (symbol-1 (string->symbol (list->string node-1))) )
  (if (last-char-eq-0? node)
      (member (list symbol-1) list-of-nodes)
      (member (list symbol-0) list-of-nodes)))

;;
;; (GET-NOT-STUCK-AT-0-NODES list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form ((AX0) BO1 (BO0)).
;;
;; -- GET-NOT-STUCK-AT-0-NODES gets the initial LIST-OF-NODES and
;; calls GET-NOT-STUCK-AT-0-NODES-1 passing it two copies of the
;; LIST-OF-NODES.
;; -- This was done to make the call less awkward from the
;; INTERPRETATION modules.

(define (get-not-stuck-at-0-nodes list-of-nodes)
  (get-not-stuck-at-0-nodes-1 list-of-nodes list-of-nodes) )

;;
;; (GET-NOT-STUCK-AT-0-NODES-1 list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form ((AX0) BO1 (BO0)).
;;   initial-list-of-nodes - a list of the same form as
;;                         list-of-nodes.
;;
;; -- GET-NOT-STUCK-AT-0-NODES-1 scans the LIST-OF-NODES to find a

```

```

;; symbol is found that is in a sublist. Not-stuck-at-faults are
;; those variables enclosed in a sublist.
;; -- Once a sublist is found at the top-level of LIST-OF-NODES, it is
;; checked to determine if the LAST-CHAR-EQ-0? in the symbol
;; enclosed in that sublist. If so, then this signifies a
;; not-stuck-at-0 condition.
;; -- In the case of a not-stuck-at-0 assertion, there does not exist
;; a stuck-at-1 variable. Otherwise, it would be designated a
;; stuck-at-1-node. Therefore, a check is made to insure that
;; there does not EXISTS-STUCK-AT-PAIR?
;; -- In the given example, there exists one not-stuck-at-0 case,
;; AX is not-stuck-at-0. REMOVE-LAST-CHAR-FROM-SYMBOL removes the
;; extraneous character from AX0.
;; -- The INITIAL-LIST-OF-NODES is scanned to find the not-stuck-at-0
;; variable in symbol form. The INITIAL-LIST-OF-NODES is carried
;; along because when checking for the nonexistence of the
;; complementary stuck-at-1 variable, it may occur either before or
;; after the not-stuck-at-0 variable in the list.

(define (get-not-stuck-at-0-nodes-1 list-of-nodes
                                     initial-list-of-nodes)
  (cond ( (null? list-of-nodes)
          '() )
        ( (not (symbol? (car list-of-nodes)))
          (if (and (last-char-eq-0? (caar list-of-nodes))
                    (not (exists-stuck-at-pair?
                           (caar list-of-nodes)
                           initial-list-of-nodes)))
                (cons (remove-last-char-from-symbol
                       (caar list-of-nodes))
                      (get-not-stuck-at-0-nodes-1
                       (cdr list-of-nodes)
                       initial-list-of-nodes)))
            (get-not-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                         initial-list-of-nodes)) )
        ( else
          (get-not-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                      initial-list-of-nodes) )))

;; (GET-NOT-STUCK-AT-1-NODES list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form ((AX1) B01 (B00)).
;;
;; -- GET-NOT-STUCK-AT-1-NODES gets the initial LIST-OF-NODES and
;; calls GET-NOT-STUCK-AT-1-NODES-1 passing it two copies of the
;; LIST-OF-NODES.
;; -- This was done to make the call less awkward from the
;; INTERPRETATION modules.

```

```

(define (get-not-stuck-at-1-nodes list-of-nodes)
  (get-not-stuck-at-1-nodes-1 list-of-nodes list-of-nodes) )

;; (GET-NOT-STUCK-AT-1-NODES-1 list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form ((AX1) B01 (B00)).
;;   initial-list-of-nodes - a list of the same form as
;;                         list-of-nodes.
;;
;; -- GET-NOT-STUCK-AT-1-NODES-1 scans the LIST-OF-NODES to find a
;; symbol is found that is in a sublist. Not-stuck-at-faults are
;; those variables enclosed in a sublist.
;; -- Once a sublist is found at the top-level of LIST-OF-NODES, it is
;; checked to determine if the LAST-CHAR-EQ-0? is not true of the
;; symbol enclosed in that sublist. If untrue, then this signifies
;; a not-stuck-at-1 condition.
;; -- In the case of a not-stuck-at-1 assertion, there does not exist
;; a stuck-at-0 variable. Otherwise, it would be designated a
;; stuck-at-0-node. Therefore, a check is made to insure that
;; there does not EXISTS-STUCK-AT-PAIR?
;; -- In the given example, there exists one not-stuck-at-1 case, AX
;; is not-stuck-at-1. REMOVE-LAST-CHAR-FROM-SYMBOL removes the
;; extraneous character from AX1.
;; -- The INITIAL-LIST-OF-NODES is scanned to find the not-stuck-at-1
;; variable in symbol form. The INITIAL-LIST-OF-NODES is carried
;; along because when checking for the nonexistence of the
;; complementary stuck-at-0 variable, it may occur either before or
;; after the not-stuck-at-1 variable in the list.

(define (get-not-stuck-at-1-nodes-1 list-of-nodes
                                      initial-list-of-nodes)
  (cond ( (null? list-of-nodes)
          '())
        ( (not (symbol? (car list-of-nodes)))
          (if (and (not (last-char-eq-0? (caar list-of-nodes)))
                    (not (exists-stuck-at-pair?
                           (caar list-of-nodes)
                           initial-list-of-nodes)))
               (cons (remove-last-char-from-symbol
                      (caar list-of-nodes))
                     (get-not-stuck-at-1-nodes-1
                      (cdr list-of-nodes)
                      initial-list-of-nodes)))
               (get-not-stuck-at-1-nodes-1 (cdr list-of-nodes)
                                           initial-list-of-nodes)) )
        ( else
          (get-not-stuck-at-1-nodes-1 (cdr list-of-nodes)

```

```

initial-list-of-nodes) )))

;; (EXISTS-STUCK-AT-PAIR? node list-of-nodes)
;;
;; Parameters:
;;   node - a symbol of either the form AX0 or BX1
;;   list-of-nodes - a list of the form ((AX0) (B01) B00).
;;
;; -- EXISTS-STUCK-AT-PAIR? takes the input NODE and creates the
;; opposite symbol regardless of the input form. Then depending on
;; the last character of the NODE, the appropriate symbol is used
;; to see whether it is in the LIST-OF-NODES.
;; -- For example, if AX0 were the input NODE, then a test would be
;; made to determine whether AX1 is a member of LIST-OF-NODES.
;; -- IF B01 were the NODE, then a test would be made to determine
;; whether B00 was a member of LIST-OF-NODES.

(define (exists-stuck-at-pair? node list-of-nodes)
  (let* ( (node-less-last-char (remove-last-char-from-symbol node))
          (node-1 (string->list
                        (symbol->string node-less-last-char)))
          (node-0 (append node-1
                        (string->list (number->string 0 '(int))))))
    (node-1 (append node-1
                    (string->list (number->string 1 '(int))))))
    (symbol-0 (string->symbol (list->string node-0)))
    (symbol-1 (string->symbol (list->string node-1))) )
  (if (last-char-eq-0? node)
      (member symbol-1 list-of-nodes)
      (member symbol-0 list-of-nodes)))))

;; (GET-ONLY-STUCK-AT-0-NODES list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form (AX0 B00 (B01)).
;;
;; -- GET-ONLY-STUCK-AT-0-NODES gets the initial LIST-OF-NODES and
;; calls GET-ONLY-STUCK-AT-0-NODES-1 passing it two copies of the
;; LIST-OF-NODES.
;; -- This was done to make the call less awkward from the
;; INTERPRETATION modules.

(define (get-only-stuck-at-0-nodes list-of-nodes)
  (get-only-stuck-at-0-nodes-1 list-of-nodes list-of-nodes) )

```

```

;; (GET-ONLY-STUCK-AT-0-NODES-1 list-of-nodes initial-list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form (AX0 B00 (B01)).
;;   initial-list-of-nodes - a list of the same form as
;;                         list-of-nodes.
;;
;; -- GET-ONLY-STUCK-AT-0-NODES-1 scans the LIST-OF-NODES to see if
;;   a symbol is found that is not in a sublist. Stuck-at-faults are
;;   those variables not enclosed in a sublist.
;; -- Once a symbol is found at the top-level of LIST-OF-NODES, it is
;;   checked to determine if the LAST-CHAR-EQ-0? If so, then this
;;   signifies a stuck-at-0 fault.
;; -- In this condition, there does not exist a complementary
;;   not-stuck-at-1 variable. Thus, if EXISTS-NOT-STUCK-AT-PAIR? not
;;   true, then the given symbol is called "only" stuck-at-1. In the
;;   given example, there exists one only-stuck-at-0 case, AX. Thus,
;;   the returned list would be (AX). REMOVE-LAST-CHAR-FROM-SYMBOL
;;   removes the extraneous character from the AX0 symbol.
;; -- The INITIAL-LIST-OF-NODES is scanned to find the stuck-at-0
;;   variable in symbol form. The INITIAL-LIST-OF-NODES is carried
;;   along because when checking for the non-existence of the
;;   complementary stuck-at-1 variable, it may occur either before or
;;   after the stuck-at-0 variable in the list.

(define (get-only-stuck-at-0-nodes-1 list-of-nodes initial-list-of-nodes)
  (cond ( (null? list-of-nodes)
          '() )
        ( (symbol? (car list-of-nodes))
          (if (and (last-char-eq-0? (car list-of-nodes))
                    (not (exists-not-stuck-at-pair?
                          (car list-of-nodes)
                          initial-list-of-nodes)))
                (cons (remove-last-char-from-symbol
                      (car list-of-nodes))
                      (get-only-stuck-at-0-nodes-1
                        (cdr list-of-nodes)
                        initial-list-of-nodes))
                (get-only-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                              initial-list-of-nodes)) . )
          ( else
            (get-only-stuck-at-0-nodes-1 (cdr list-of-nodes)
                                         initial-list-of-nodes) )))

;;
;; (GET-ONLY-STUCK-AT-1-NODES list-of-nodes)
;;
;; Parameters:
;;   list-of-nodes - A list of nodes of the form (AX1 B00 (B01)).

```

```

;;
;; -- GET-ONLY-STUCK-AT-1-NODES gets the initial LIST-OF-NODES and
;; calls GET-ONLY-STUCK-AT-1-NODES-1 passing it two copies of the
;; LIST-OF-NODES.
;; -- This was done to make the call less awkward from the
;; INTERPRETATION modules.

(define (get-only-stuck-at-1-nodes list-of-nodes)
  (get-only-stuck-at-1-nodes-1 list-of-nodes list-of-nodes) )

;; (GET-ONLY-STUCK-AT-1-NODES-1 list-of-nodes initial-list-of-nodes)

;;
;; Parameters:
;; list-of-nodes - A list of nodes of the form (AX1 B00 (B01)).
;; initial-list-of-nodes - a list of the same form as
;; list-of-nodes.

;;
;; -- GET-ONLY-STUCK-AT-1-NODES-1 scans the LIST-OF-NODES to see a
;; symbol is found that is not in a sublist. Stuck-at-faults are
;; those variables not enclosed in a sublist.
;; -- Once a symbol is found at the top-level of LIST-OF-NODES, it is
;; checked to determine if LAST-CHAR-EQ-0? is not true. If so,
;; then this signifies a stuck-at-1 fault.
;; -- In this condition, there does not exist a complementary
;; not-stuck-at-0 variable. Thus, if EXISTS-NOT-STUCK-AT-PAIR? not
;; true, then the given symbol is called "only" stuck-at-1. In the
;; given example, there exists one only-stuck-at-1 case, AX. Thus,
;; the returned list would be (AX). REMOVE-LAST-CHAR-FROM-SYMBOL
;; removes the extraneous character from the AX1 symbol.
;; -- The INITIAL-LIST-OF-NODES is scanned to find the stuck-at-1
;; variable in symbol form. The INITIAL-LIST-OF-NODES is carried
;; along because when checking for the non-existence of the
;; complementary stuck-at-0 variable, it may occur either before or
;; after the stuck-at-1 variable in the list.

(define (get-only-stuck-at-1-nodes-1 list-of-nodes
                                      initial-list-of-nodes)
  (cond ( (null? list-of-nodes)
          '())
        ( (symbol? (car list-of-nodes))
          (if (and (not (last-char-eq-0? (car list-of-nodes)))
                    (not (exists-not-stuck-at-pair?
                           (car list-of-nodes)
                           initial-list-of-nodes)))
                (cons (remove-last-char-from-symbol
                           (car list-of-nodes))
                      (get-only-stuck-at-1-nodes-1
                        (cdr list-of-nodes)
                        initial-list-of-nodes)))
            )
        )
    )

```

```

        (get-only-stuck-at-1-nodes-1 (cdr list-of-nodes)
                                      initial-list-of-nodes)) )
( else
  (get-only-stuck-at-1-nodes-1 (cdr list-of-nodes)
                                      initial-list-of-nodes) )))

;; (LAST-CHAR-EQ-0? symbol)
;;
;; Parameter:
;;   symbol - an arbitrary symbol
;;
;; -- LAST-CHAR-EQ-0? decomposes the symbol into a list of characters.
;;   GET-LAST-ELT takes this list of characters and returns the
;;   LAST-CHAR.
;; -- If LAST-CHAR equals 0, then #T is returned; '() otherwise.

(define (last-char-eq-0? symbol)
  (let* ( (symbol-1 (string->list (symbol->string symbol)))
          (last-char (get-last-elt symbol-1)) )
    (equal? last-char '(#\0)) ))


;; (SHOW-INPUT-NODES list-of-nodes option)
;;
;; Parameters:
;;   list-of-nodes - a list of the nodes to be output
;;   option - a symbol which designates what type of fault (or
;;           normal) is associated with the given LIST-OF-NODES
;;
;; -- SHOW-INPUT-NODES takes the first node off the LIST-OF-NODES and
;;   outputs this node with the status dependent on the OPTION input
;;   to the procedure.
;; -- SHOW-INPUT-NODES calls itself recursively until all nodes in the
;;   original LIST-OF-NODES are exhausted.

(define (show-input-nodes list-of-nodes option)
  (if (null? list-of-nodes)
      '()
      (begin
        (display "      Input node ")
        (display (remove-last-char-from-symbol (car list-of-nodes)))
        (cond ( (equal? option 'normal)
                 (writeln " is normal.") )
              ( (equal? option 'stuck-at-0)
                 (writeln " is stuck-at-0.") )
              ( (equal? option 'stuck-at-1)
                 (writeln " is stuck-at-1.") )
              ( (equal? option 'not-stuck-at-0)

```

```

        (writeln " is not stuck-at-0." ) )
( (equal? option 'not-stuck-at-1)
  (writeln " is not stuck-at-1." ) )
( (equal? option 'only-stuck-at-0)
  (writeln " is stuck-at-0." ) )
( (equal? option 'only-stuck-at-1)
  (writeln " is stuck-at-1." ) ) )
(show-input-nodes (cdr list-of-nodes) option)))))

;; (SHOW-FANOUT-NODES list-of-nodes option prefix-list)
;;
;; Parameters:
;;   list-of-nodes - a list of nodes to be output
;;   option - a symbol which designates what type of fault (or
;;           normal) is associated with the given LIST-OF-NODES
;;   prefix-list - The prefix list of the input circuit; this list
;;                 was modified to MAKE-UNIQUE-FANOUTS of each of the
;;                 fanout nodes. This is necessary to distinguish
;;                 the fanouts and associate them with the list of
;;                 fanout nodes.
;;
;; -- SHOW-FANOUT-NODES iteratively outputs each of the elements of
;; LIST-OF-NODES displaying in sequence the node, the gate
;; associated with that node, and then the status of that node.
;; -- The PREFIX-LIST is of the form:
;;
;;   ((EQ E- (NOT (* AX B0)))
;;    (EQ F- (NOT B1))
;;    (EQ Z- (NOT (* E- F-))))
;;
;; -- GET-GATE returns the EQUATION associated with the fanout node.
;; For example, if node B0 were to be displayed, then the EQUATION
;; returned by GET-GATE would be (EQ E- (NOT (* AX B0))). This
;; equation is displayed by SHOW-EQUATION-C.
;; -- After the gate is displayed, the status of the checkpoint is
;; displayed. This status is dependent on the OPTION which is the
;; type of fault associated with the given LIST-OF-NODES.
;; -- SHOW-FANOUT-NODES calls itself recursively until all of the
;; LIST-OF-NODES in the original list have been displayed with the
;; appropriate gate.

(define (show-fanout-nodes list-of-nodes option prefix-list)
  (if (null? list-of-nodes)
      '()
      (let ( (equation (get-gate (car list-of-nodes) prefix-list)) )
        (display "    Node ")
        (display (remove-last-char-from-symbol (car list-of-nodes)))
        (display " of gate: ")
        (show-equation-c equation)

```

```

(cond ( (equal? option 'normal)
         (writeln " is normal." ) )
      ( (equal? option 'stuck-at-0)
         (writeln " is stuck-at-0." ) )
      ( (equal? option 'stuck-at-1)
         (writeln " is stuck-at-1." ) )
      ( (equal? option 'not-stuck-at-0)
         (writeln " is not stuck-at-0." ) )
      ( (equal? option 'not-stuck-at-1)
         (writeln " is not stuck-at-1." ) )
      ( (equal? option 'only-stuck-at-0)
         (writeln " is stuck-at-0." ) )
      ( (equal? option 'only-stuck-at-1)
         (writeln " is stuck-at-1." ) ) )
(show-fanout-nodes (cdr list-of-nodes)
                    option
                    prefix-list) )))

;; (SHOW-EQUATION-C equation)
;;
;; Parameters:
;;   equation - a list of the form (EQ E- (NOT (* AX BO)))
;;
;; -- SHOW-EQUATION-C displays the above equation in the form
;;   E = A * B.
;; -- First, the output node for the gate is displayed followed by an
;;   equals sign. Then either SHOW-NEGATED-EQUATION, or SHOW-FORMULA
;;   are called depending on whether the gate is of the NEGATED
;;   variety, i.e. NAND.
;; -- The difference between this procedure and SHOW-EQUATION is that
;;   SHOW-EQUATION outputs a NEWLINE after the equation. This
;;   procedure does not.

(define (show-equation-c equation)
  (let* ( (output (cadr equation))
          (input (caddr equation)) )
    (display (remove-last-char-from-symbol output))
    (display " = ")
    (if (equal? (car input) 'NOT)
        (show-negated-equation (cadr input))
        (show-formula input)))))

;; (GET-GATE node prefix-list)
;;
;; Parameters:
;;   node - a node which is to be matched with the right side
;;          of an equation in prefix-form to get specific gate

```

```

;;      that a fanout node is associated with
;; prefix-list - a list of the form:
;;
;;      ((EQ E- (NOT (* AX BO)))
;;       (EQ F- (NOT BI))
;;       (EQ Z- (NOT (* E- F-))))
;;
;; -- GET-GATE returns the single equation that node is associated
;; with.
;; -- If the NODE were BO, then the equation (EQ E- (NOT (* AX BO)))
;; would be returned. ON-RIGHT-SIDE? is used to determine whether
;; the NODE is on the right side of the first equation in the list
;; of equations.
;; -- GET-GATE calls itself recursively until the appropriate equation
;; is found.

(define (get-gate node prefix-list)
  (let ( (right-side-of-first-list (caddr prefix-list)) )
    (if (on-right-side? node right-side-of-first-list)
        (car prefix-list)
        (get-gate node (cdr prefix-list)))))

;;
;; (SHOW-FORMULA formula)
;;
;; Parameters:
;;   formula - a formula of the form (* E- F-)
;;
;; -- SHOW-FORMULA displays the second element of the list which will
;; only be a symbol. REMOVE-LAST-CHAR-FROM-SYMBOL gets rid of
;; the "-" character in this case.
;; -- Following the output of the symbol, the FIRST-ELT of the list
;; is displayed. This will always be a token symbol such as
;; * or +.
;; -- SHOW-FORMULA is then called recursively to display the REST of
;; the formula. This is necessary because for a four-input
;; and-gate the input FORMULA would be of the form:
;; (* A- (* B- (* C- D-))). SHOW-FORMULA would display this as:
;;   A * B * C * D.

(define (show-formula formula)
  (cond ( (null? formula)
           '() )
        ( (symbol? formula)
           (display (remove-last-char-from-symbol formula)) )
        ( else
           (let ( (first-elt (car formula))
                 (second-elt (cadr formula))
                 (rest       (caddr formula)) )
             (display (remove-last-char-from-symbol second-elt))))
```

```

(display " ")
(display first-elt)
(display " ")
(show-formula rest)) )))

;; (SHOW-NEGATED-EQUATION formula)
;;
;; Parameters:
;;   formula - a formula of the form (* E- F-)
;;
;; -- SHOW-NEGATED-EQUATION is called by SHOW-EQUATION and
;;   SHOW-EQUATION-C to show an equation such as that represented by
;;   a NAND, NOR, or NOT gate.
;; -- SHOW-NEGATED-EQUATION calls SHOW-FORMULA to display the interior
;;   formula, surrounding it with the "(" formula ")" symbols.

(define (show-negated-equation formula)
  (display "(")
  (show-formula formula)
  (display ")")) )

;; (PAUSE)
;;
;; -- PAUSE waits in a loop until the carriage return is pressed.

(define (pause)
  (if (equal? (read-char) #\return)
      '()
      (pause)))

;; (DISPLAY-SYSTEM-METRICS inputs no-of-tests)
;;
;; Parameters:
;;   inputs - the number of inputs of the circuit
;;   no-of-tests - the number of input-output tests that were
;;                 conducted before all information was determined
;;
;; -- DISPLAY-SYSTEM-METRICS prints out messages about the number of
;;   tests that were run as opposed to the number of tests that it
;;   was possible to have run.
;; -- NO-OF-POSSIBLE-TESTS take the list of INPUTS nodes and returns
;;   the number of possible tests that could have been run
;;   (2 to the n).
;; -- The Performance ratio output is the ratio of the NO-OF-TESTS
;;   with respect to the number of possible tests that could have

```

```

;;      been run.

(define (display-system-metrics inputs no-of-tests)

  (newline)
  (writeln "System Performance Metrics:")
  (newline)
  '(writeln "    The number of tests run was: " no-of-tests)
  (display "    The number of possible tests was: ")
  (writeln (no-of-possible-tests inputs))
  (newline)
  (display "    The performance ratio is: ")
  (writeln (/ no-of-tests (no-of-possible-tests inputs)))
  (newline)
  (newline) )

;; (NO-OF-POSSIBLE-TESTS inputs)
;;
;; Parameters:
;;   inputs - the list of inputs to the circuit
;;
;; -- NO-OF-POSSIBLE-TESTS takes the length of the INPUTS list to
;; determine n.  The number of possible tests that can be run on
;; the circuit is 2 to the nth power.  This is the number that is
;; returned.

(define (no-of-possible-tests inputs)
  (expt 2 (length inputs)) )

```

```

;;;;;;
;;
;:;
;:;           Filename: boolean.s
;:;
;: These procedures are used for processing Boolean
;: sum-of-products (SOP) formulas. The representation assumed
;: for an SOP an formula is a list of representations of the
;: terms in the formula. A term is represented as a list of
;: representations of the literals in the term. A literal
;: X is represented by the symbol X (X may be an arbitrarily
;: complex symbol); a literal X' is represented by the list (X).
;: Thus the Boolean formula  $ax + bc'x'y + y'z' + dz + xy'$  is
;: represented by the list
;:   ( (a x) (b (c) (x) y) ((y) (z)) (d z) (x (y)) ) .
;: Let F1,...,Fn, G1,...,Gn be representations for SOP
;: formulas. Then the system
;:   F1 = G1
;:   F2 = G2
;:   ...
;:   Fn = Gn
;: is represented by the list
;:   ( (eq F1 G1) (eq F2 G2) ... (eq Fn Gn) ) .
;: A logical inclusion  $F \leq G$  (i.e., F implies G) appearing in
;: a system is represented by the sub-list (le F G), read
;: "F is less than or equal to G." Thus the system
;:   a'b'c'      =  x'yz
;:   ab + ac    =  x'y' + y'z'
;:   a'b + b'c  =<  xy + x'z
;:   ac + bc + a'c' =< x' + yz
;:   ab'        =<  x' + z'
;: is represented by the list
;:   ( (eq (((a) (b) (c)))      (((x) y (z)))) )
;:     (eq ((a b) (a c))      (((x) (y)) ((y) (z)))) )
;:     (le (((a) b) ((b) c))    ((x y) ((x) z))) )
;:     (le ((a c) (b c) ((a) (c))) (((x) (y z)))) )
;:     (le ((a (b)))          (((x) ((z)))) ) )
;:;
;: These procedures were written by Dr. Frank M. Brown, Professor of
;: of Electrical Engineering at the Air Force Institute of Technology
;:;;
;;;;;;
;;;;
;:;           BASIC BOOLEAN OPERATORS ;;;;;
;:;
;: (add f1 f2) returns the logical OR, in SOP form, of two SOP
;: formulas. Elementary simplification-housekeeping is performed
;: on the result.
;:;
(define (add f1 f2)
  (cond ( (null? f1) f2)
        ( (absorbed? (car f1) f2)

```

```

        (add (cdr f1) f2) )
( else
  (add (cdr f1)
    (cons (car f1)
      (remove-supersets (car f1) f2) )))))

```

```

(define (remove-supersets term f)
(cond ( (null? f) nil)
  ( (subset? term (car f))
    (remove-supersets term (cdr f)) )
  ( else
    (cons (car f)
      (remove-supersets term (cdr f)) )))))

```

;; (mult f g) returns the logical AND, in SOP form, of the
 ;; SOP formulas f and g. The recursive rule
 ;; $f * g = x'(f/x' * g/x') + x(f/x * g/x)$
 ;; is implemented, where x is any argument appearing in f.

```

(define (mult f g)
(cond ((null? f) nil)
  ((null? g) nil)
  ((member nil f) g)
  ((member nil g) f)
  (else
    (let* ((arg (first-arg f))
      (narg (bar arg))
      (f0 (divide f narg))
      (f1 (divide f arg))
      (g0 (divide g narg))
      (g1 (divide g arg))
      (product0 (mult f0 g0))
      (product1 (mult f1 g1)) )
      (append (prefix narg product0)
        (prefix arg product1) )))))

```

;; (XOR F G) returns the logical EXCLUSIVE-OR, in SOP form, of
 ;; the SOP formulas F and G. The recursive rule
 ;; $f \text{ XOR } g = x'(f/x' \text{ XOR } g/x') + x(f/x \text{ XOR } g/x)$
 ;; is implemented, where x is any argument appearing in f.

```

(define (xor f g)
(cond ((null? f) g)
  ((null? g) f)
  ((member nil f) (complement g))

```

```

((member nil g) (complement f))
(else
  (let* ((arg (first-arg f))
         (narg (bar arg))
         (f0 (divide f narg))
         (f1 (divide f arg))
         (g0 (divide g narg))
         (g1 (divide g arg))
         (xor0 (xor f0 g0))
         (xor1 (xor f1 g1)) )
    (append (prefix narg xor0)
            (prefix arg xor1) )))))

(define (xnor f g)
  (cond ( (null? f) (complement g) )
        ( (null? g) (complement f) )
        ( (member nil f) g)
        ( (member nil g) f)
        ( else
          (let* ( (arg (first-arg f))
                  (narg (bar arg))
                  (f0 (divide f narg))
                  (f1 (divide f arg))
                  (g0 (divide g narg))
                  (g1 (divide g arg))
                  (xnor0 (xnor f0 g0))
                  (xnor1 (xnor f1 g1)) )
            (append (prefix narg xnor0)
                    (prefix arg xnor1)) )))))

;; (complement f) returns the complement, in SOP form, of the SOP
;; formula f. The recursive rule
;;           f' = x'(f/x')' + x(f/x)'
;; is implemented, where x is any argument of f. Thus each term
;; of the formula for f' will contain x' or x as a literal.

(define (complement f)
  (cond ((null? f) (list nil))
        ((member nil f) nil)
        (else
          (let* ((arg (first-arg f))
                 (narg (bar arg))
                 (f0 (divide f narg))
                 (f1 (divide f arg))
                 (comp0 (complement f0))
                 (comp1 (complement f1)) )
            (append (prefix narg comp0)
                    (prefix arg comp1))))
```

```

        (prefix arg  comp1 )))))

;; (BCF F) returns the Blake canonical form of the sum-of-products
;; formula F. The interior list-format is returned.
;; Both procedures are based on the relation
;;
;;      BCF(f) = ABS([x' + BCF(f/x)] # [x + BCF(f/x')])    (1)
;;
;; where
;;      -- ABS is an operator which removes absorbed terms;
;;      -- f/u denotes the quotient of f with respect to u, i.e.,
;;          the result of making the substitution u = 1 in f; and
;;      -- # is the "mu-product" operator, indicating explicit
;;          term-by-term cross-multiplication.
;;
;; When (1) is multiplied out, the result is
;;
;;      BCF(f) = ABS(x'BCF0 + x BCF1 + PROD)           (2)
;;
;; where BCF0 denotes BCF(f/x'), BCF1 denotes BCF(f/x ) and
;; PROD denotes BCF0 # BCF1. The only absorptions possible
;; are (a) those within PROD and (b) absorptions of terms in
;; x'BCF0 or x BCF1 by terms in PROD.

(define (bcf f)
  (cond ( (null? f) f)
        ( (null? (cdr f)) f)
        ( (member nil f) (list nil))
        ( else
            (let ((arg (opposed-arg f)))
              (cond ( (null? arg)
                        (unabsorb f) )
                    ( else
                        (let* ( (narg (bar arg))
                                (f0  (divide f narg))
                                (f1  (divide f arg))
                                (bcf0 (bcf f0))
                                (bcf1 (bcf f1))
                                (prod (muprod bcf0 bcf1))
                                (absprod (unabsorb prod))
                                (nf0  (absorb-rel bcf0 absprod))
                                (nf1  (absorb-rel bcf1 absprod)) )
                            (append (prefix narg nf0)
                                    (prefix arg nf1)
                                    absprod))))))))

```

;; (UMABSORB F) returns a subformula of F that contains no terms

```

;; that are absorbed by other terms in the subformula.

(define (unabsorb f)
  (cond ( (null? f) nil)
        ( else
          (insert-abs (car f)
                      (unabsorb (cdr f)) ))))

(define (insert-abs term f)           ; Insert a term into an
  (cond ( (null? f) (list term))      ; absorbed-out formula,
        ( (subset? term (car f))       ; and carry out all ab-
          (insert-abs term (cdr f)) )  ; sorptions on the result.
        ( (subset? (car f) term)
          f )
        ( else
          (cons (car f)
                 (insert-abs term (cdr f)) ))))

;; (ABSORB-REL F G) returns those terms of the SOP formula F
;; that are not absorbed by any term of the SOP formula G.

(define (absorb-rel f g)
  (cond ( (null? g) f)
        ( else
          (absorb-rel (term-absorb f (car g)) (cdr g)) )))

(define (term-absorb f term)
  (cond ( (null? f) nil)
        ( (subset? term (car f))
          (term-absorb (cdr f) term) )
        ( else
          (cons (car f) (term-absorb (cdr f) term)) )))

;; The mu-product of two SOP formulas is the term-by-term
;; product.

(define (muprod f g)
  (cond ( (null? f) nil)
        ( else
          (append (muprod-tf (car f) g)
                  (muprod (cdr f) g) ))))

```

```

(define (muprod-tf term f)
  (cond ( (null? term) f)
        ( (null? f)    nil)
        ( else
          (let ( (prod (muprod-tt term (car f))))
                (cond ( (equal? prod 0)
                          (muprod-tf term (cdr f)) )
                      (else
                        (cons prod
                              (muprod-tf term (cdr f)) )))))))))

```

```

(define (muprod-tt t1 t2)
  (cond ( (null? t1) t2)
        ( (member (bar (car t1)) t2)
          0 )
        ( else
          (let ( (rest (muprod-tt (cdr t1) t2)))
                (cond ( (equal? rest 0)
                          0 )
                      ( (member (car t1) t2)
                        rest )
                      ( else
                        (cons (car t1) rest) ))))))))

```

;; (ABSORBED? TERM F) is a predicate returning TRUE in case
 ;; TERM is absorbed by some term of the SOP formula F.

```

(define (absorbed? term f)
  (cond ((null? f) nil)
        ((subset? (car f) term) true)
        (else
          (absorbed? term (cdr f) ))))

```

;; (ECON F TERM) returns the conjunctive eliminant of F with
 ;; respect to the arguments in TERM.

```

(define (econ f term)
  (simplify (econ2 f term)) )

```

```

(define (econ2 f term)
  (cond ((null? f) f)
        ((member nil f) (list nil)))

```

```

((null? term) f)
(else
  (let* ((arg (car term))
         (part (partition f arg))
         (p (car part))
         (q (cadr part))
         (r (caddr part))
         (prod (mult p q)) )
    (econ2 (append prod r) (cdr term)) ))))

;;; (EDIS F TERM) returns the disjunctive eliminant of F with
;;; respect to term

(define (edis f term)
  (cond ( (null? term) f)
        ( else
          (edis (replace-by-one f (car term))
                (cdr term) ))))

(define (replace-by-one f x)
  (cond ( (null? f) nil)
        ( else
          (cons (replace-term (car f) x)
                (replace-by-one (cdr f) x) ))))

(define (replace-term term x)
  (cond ( (null? term) nil)
        ( (or (equal? (car term) x)
              (equal? (car term) (bar x)) )
            (cdr term) )
        ( else
          (cons (car term)
                (replace-term (cdr term) x) ))))

;; This procedure is a slight modification of BCF-ITER, which
;; implements the method of iterated consensus.
;; It differs from BCF-ITER only in the procedure "consensus."
;; The idea here is to generate only consensus-terms which
;; absorb at least one of their parents. The principal utility
;; of this procedure is to clean up functions like MULT and
;; ECON (which uses MULT) whose recursive definition causes
;; them to produce large numbers of terms that differ in only
;; one literal.

```

```

(define (simplify f)
  (unabsorb
    (simplify2 nil f) ))


(define (simplify2 left right)
  (cond ( (null? right) left)
        ( (null? left)
          (simplify2 (list (car right))
                     (cdr right) )))
        ( (absorbed? (car right) left)
          (simplify2 left
                     (cdr right) )))
        ( else
          (let ((newcons (sweep (car right) left)))
            (cond ( (equal? newcons '())
                     '())
                  ( (equal? (car newcons) 'drop-term)
                    (simplify2 left
                               (append (cadr newcons)
                                      (cdr right) )))
                  ( else
                    (simplify2 (cons (car right) left)
                               (append (cadr newcons)
                                      (cdr right) )))))))))

```



```

(define (sweep term f)
  (sweep2 term nil f) )


```



```

(define (sweep2 term acc partf)
  (cond ( (null? partf) (list 'ok acc))
        ( else
          (let ((consens (consensus term (car partf))))
            (cond ( (null? consens)
                     (sweep2 term acc (cdr partf)) )
                  ( (equal? consens '())
                     '())
                  ( (subset? consens term)
                    (list 'drop-term
                          (cons consens acc) ))
                  ( else
                    (sweep2 term (cons consens acc)
                           (cdr partf) )))))))

```

; ; p and q are terms. The consensus of p and q is returned only
; ; if the consensus absorbs at least one of its parents.

```
(define (consensus p q)
  (let ( (consens (consensus2 0 p q)) )
    (cond ( (equal? consens 'no-dice)
            '())
          ( (null? consens)
            '())
          ( (or (subset? consens p)
                 (subset? consens q) )
            consens )
          ( else
            '() ))))

(define (consensus2 count p acc)
  (cond ( (= count 0)
            (cond ( (null? p)
                    'no-dice )
                  ( (member (bar (car p)) acc)
                    (consensus2 1 (cdr p)
                               (remove (bar (car p)) acc) )))
                  ( (member (car p) acc)
                    (consensus2 0 (cdr p) acc) )
                  ( else
                    (consensus2 0 (cdr p) (cons (car p) acc)) )))
            ( else
              (cond ( (null? p)
                      acc )
                    ( (member (bar (car p)) acc)
                      'no-dice )
                    ( (member (car p) acc)
                      (consensus2 1 (cdr p) acc) )
                    ( else
                      (consensus2 1 (cdr p) (cons (car p) acc)) )))))
        ( else
          (cond ( (null? p)
                  acc )
                ( (member (bar (car p)) acc)
                  'no-dice )
                ( (member (car p) acc)
                  (consensus2 1 (cdr p) acc) )
                ( else
                  (consensus2 1 (cdr p) (cons (car p) acc)) )))))

(define (divide f x)
  (cond ((null? f) nil)
        ((member (bar x) (car f))
         (divide (cdr f) x) )
        (else
         (cons (remove x (car f))
               (divide (cdr f) x) ))))
```

```

(define (partition f x)
  (let* ( (arg (debar x))
          (narg (bar arg)) )
    (partition1 f arg narg) ))


(define (partition1 f arg narg)
  (cond ((null? f) (list nil nil nil))
        ((member narg (car f))
         (let ((next (partition1 (cdr f) arg narg)))
           (cons (cons (remove narg (car f))
                        (car next) )
                  (cdr next) )))
        ((member arg (car f))
         (let ((next (partition1 (cdr f) arg narg)))
           (list (car next)
                 (cons (remove arg (car f))
                       (cadr next) )
                 (caddr next) ))))
        (else
         (let ((next (partition1 (cdr f) arg narg)))
           (list (car next)
                 (cadr next)
                 (cons (car f)
                       (caddr next) ))))))))

;; (SUBSET? T1 T2) returns TRUE in case every member of the list
;; T1 is also a member of the list T2.

(define (subset? t1 t2)
  (cond ((null? t1) true)
        ((and (member (car t1) t2)
              (subset? (cdr t1) t2) ))
        (else nil) ))


(define (union list1 list2) ; Returns the union of
  (cond ( (null? list1) list2) ; LIST1 and LIST2, assum-
        ( (member (car list1) list2) ; ing that LIST2 has no
          (union (cdr list1) list2) ) ; duplicate elements.
        ( else
          (union (cdr list1) (cons (car list1) list2)) )))

(define (flatten lst)
  (cond ( (null? lst) nil)

```

```

( (atom? (car lst))
  (cons (car lst)
        (flatten (cdr lst)) ))
( else
  (append (flatten (car lst))
          (flatten (cdr lst)) ))))

(define (debar literal)
  (cond ((atom? literal) literal)
        (else (bar literal)) ))

;; (OPPOSED-ARG F) returns an argument that is opposed in F, if
;; one exists; otherwise, it returns nil. An argument is opposed
;; in F if the argument appears uncomplemented in one term of F
;; and complemented in another.

(define (opposed-arg f)
  (cond ( (null? f) nil)
        ( (null? (cdr f)) nil)
        ( (member nil f) nil)
        ( else
          (make-letter
            (seek-opposed (get-literals f)) ))))

(define (seek-opposed args)
  (cond ( (null? args) nil)
        ( (null? (cdr args)) nil)
        ( (member (bar (car args)) (cdr args))
              (car args) )
        ( else
          (seek-opposed (cdr args)) )))

(define (make-letter literal)
  (cond ( (atom? literal) literal)
        ( else
          (bar literal) )))

;; (GET-LITERALS F) collects in a list all of the
;; literals explicit in SOP formula F. Duplicates are
;; excluded. Example:
;; [2] (get-literals '((a (b) d) ((c) (d) f) (e (f))))
```

```

;; (D (B) & F (D) (C) (F) E)

(define (get-literals f)
  (cond ((null? f) nil)
        (else
         (union (car f) (get-literals (cdr f))))))

(define (first-arg f) ; Return, uncomplemented,
  (cond ((null? f) nil) ; the first argument en-
        ((member nil f) nil) ; countered in the function
        (else ; F.
         (debar (car (car f))))))

(define (remove x lst) ; Remove one occurrence
  (cond ((null? lst) nil) ; of element X from list
        ((equal? (car lst) x) ; LST.
         (cdr lst))
        (else
         (cons (car lst)
               (remove x (cdr lst))))))

(define (prefix x f) ; Prefix each term in the
  (cond ((null? f) nil) ; formula f by the literal
        (else ; x.
         (cons (cons x (car f))
               (prefix x (cdr f))))))

(define (bar x) ; Complement (bar) a literal x. Thus (BAR TOM)
  (cond ((atom? x) (list x)) ; returns (TOM) and
        (else (car x)) )) ; (BAR (TOM)) returns TOM.

(define (common-args? term1 term2)
  (cond ((null? term2) nil)
        ((or (member (car term2) term1)
             (member (bar (car term2)) term1))
         t)
        ((common-args? term1 (cdr term2))
         t)))

```

```
(define (sort-term term)
  (cond ((null? term) nil)
        ((null? (cdr term)) term)
        (else
          (let ((sort-cdr (sort-term (cdr term))))
            (cond ((lower-literal? (car term)
                                   (car sort-cdr) )
                  (cons (car term) sort-cdr) )
                  (else
                    (cons (car sort-cdr)
                          (sort-term (cons (car term)
                                           (cdr sort-cdr)
                                         ))))))))))
```



```
(define (lower-literal? x y)
  (lower-symbol? (debar x) (debar y)) )
```



```
(define (lower-symbol? x y)
  (let ((stringx (symbol->string x))
        (stringy (symbol->string y)) )
    (cond ( (string<? stringx stringy) true)
          (else nil) )))
```

Bibliography

[Abadi 85] Abadir, M.S. and H.K. Reghbati. "LSI Testing Techniques," *Tutorial: VLSI Testing and Validation Techniques*, 6-23. Washington, D.C.: IEEE Computer Society Press, 1985.

[Abels 85] Abelson, Harold, Gerald Jay Sussman and Julie Sussman. *Structure and Interpretation of Computer Programs*. New York: McGraw-Hill, 1985.

[Abram 80] Abramovici, Miron and Melvin A. Breuer. "Multiple Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis," *IEEE Transactions on Computers*, C-29: 451-460 (June 1980).

[Agarw 81] Agarwal, Vinod K. and Andy S.F. Fung. "Multiple Fault Testing of Large Circuits by Single Fault Test Sets," *IEEE Transactions on Computers*, C-30: 855-865 (November 1981).

[Agraw 88] Agrawal, Vishwani D. and Sharad C. Seth. *Tutorial: Test Generation for VLSI Chips*. Washington, D.C.: IEEE Computer Society Press, 1988.

[Akers 74] Akers, Sheldon B. Jr. "Fault Diagnosis as a Graph Coloring Problem," *IEEE Transactions on Computers*, C-23: 706-713 (July 1974).

[Al-Ar 87a] Al-Arian, Sami A. "Testing Algorithms of Open Faults in CMOS Networks," *Conference Proceedings IEEE SOUTHEASTCON 1987*, 2. 352-359. Piscataway, NJ: IEEE, 1987.

[Al-Ar 87b] Al-Arian, Sami A. and Martin Nordens. "Fault Models of Basic CMOS Structures," *Conference Proceedings IEEE SOUTHEASTCON 1987*, 2. 360-366. Piscataway, NJ: IEEE, 1987.

[Armst 66] Armstrong, D.B. "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," *IEEE Transactions on Electronic Computers*, EC-15: 66-73 (February 1966).

[Baner 84] Banerjee, Prithviraj and Jacob A. Abraham. "Characterization and Testing of Physical Failures in MOS Logic Circuits," *IEEE Design and Test of Computers*, 1: 76-86 (August 1984).

[Bate 88] Bate, J.A. and D.M. Miller. "Exhaustive Testing of Stuck-Open Faults in CMOS Combinational Circuits," *IEE Proceedings, Pt.E*, 135: 10-16 (January 1988).

[Bearn 71] Bearson, L.W. and Chester C. Carroll. "On the Design of Minimum Length Fault Tests for Combinational Circuits," *IEEE Transactions on Computers*, C-20: 1353-1356 (November 1971).

[Blake 37] Blake, Archie. *Canonical Expressions in Boolean Algebra*, PhD Dissertation. Department of Mathematics, University of Chicago, Chicago, Illinois, 1937.

[Boole 54] Boole, George. *An Investigation of the Laws of Thought*. Originally published in 1854 by Macmillan, London. Reprinted by Dover Publications in 1958.

[Boss 71] Bossen, Douglas C. and Se Jung Hong. "Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks," *IEEE Transactions on Computers*, C-20: 1252-1257 (November 1971).

[Breue 76a] Breuer, Melvin A., Shih-Jeh Chang, and Stephen Y. H. Su. "Identification of Multiple Stuck-Type Faults in Combinational Networks," *IEEE Transactions on Computers*, C-25: 44-54 (January 1976).

[Breue 76b] Breuer, Melvin A. and Arthur D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Woodland Hills, California: Computer Science Press, 1976.

[Brown 88a] Brown, Frank M. *Boolean Reasoning with Applications in Logical Design*. Unpublished textbook. School of Engineering, Air Force Institute of Technology (AU), Wright Patterson AFB, OH 1988.

[Brown 79] Brown, Frank M. *Boolean Equations and Logical Diagnosis*. U.S. Air Force Contract F49620-79-C-0038.

[Brown 88b] Brown, Frank M. Personal Interviews. School of Engineering, Air Force Institute of Technology (AU), Wright Patterson AFB, OH 1988.

[Burge 88] Burgess, N. and others. "Physical Faults in MOS Circuits and Their Coverage by Different Fault Models," *IEE Proceedings, Pt.E*, 135: 1-9 (January 1988).

[Carro 74] Carroll, B.D., H.G. Shah, and D.M.Jones. "An Examination of Algebraic Test Generation Methods for Multiple Faults," *IEEE Transactions on Computers*, C-23: 743-745 (July 1974).

[Cerny 76] Cerny, E. "Application of a Boolean-Equation-Based Methodology to the Detection of Faults in Combinational Switching Circuits," *IEEE Computer Repository*, Report R76-84. 1976.

[Cha 79] Cha, C.W. "Multiple Fault Diagnosis in Combinational Networks," *Proceedings of the Sixteenth Design Automation Conference*. 149-155. Washington, D.C.: IEEE Computer Society Press, 1979.

[Chand 78] Chandra, Ashok K. and George Markowsky. "On the Number of Prime Implicants," *Discrete Mathematics*, 24: 7-11 (October 1978).

[Davis 85] Davis, Randall. "Diagnostic Reasoning Based on Structure and Behavior," *Tutorial: VLSI Testing and Validation Techniques*. 515-578. Washington, D.C.: IEEE Computer Society Press, 1985.

[deKle 87] deKleer, Johan and Brian C. Williams. "Diagnosing Multiple Faults," *Artificial Intelligence*, 32: 97-130 (April 1987).

[Dunha 59] Dunham, B. and R. Fridshal. "The Problem of Simplifying Logical Expressions," *The Journal of Symbolic Logic*, 24: 17-19 (March 1959).

[Dybvi 87] Dybvig, R. Kent. *The SCHEME Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[Eisen 88] Eisenberg, Michael. *Programming in Scheme*. Redwood City, California: Scientific Press, 1988.

[Fause 86] Fausett, Mark L. *A Scheme-Based System for Boolean Reasoning*, MS Thesis AFIT/GCS/86D-8. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

[Fridr 74] Fridrich, Marek and Wayne A. Davis. "Minimal Fault Tests for Combinational Networks," *IEEE Transactions on Computers*, C-23: 850-859 (August 1974).

[Fried 74] Friedman, Arthur D. "Diagnosis of Short-Circuit Faults in Combinational Circuits," *IEEE Transactions on Computers*, C-23: 746-752 (July 1974).

[Fujiw 85] Fujiwara, Hideo. *Logic Testing and Design for Testability*. Cambridge, Massachusetts: MIT Press, 1985.

[Galil 75] Galil, Zvi. *The Complexity of Resolution Procedures for Theorem Proving in the Propositional Calculus*, PhD Dissertation. Department of Computer Science, Cornell University, Ithaca, New York, 1975.

[Genes 84] Genesereth, Michael R. "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence*, 24: 411-436 (December 1984).

[Hughe 86] Hughes, Joseph L.A. and Edward J. McCluskey. "Multiple Stuck-At Fault Coverage of Single Stuck-At Fault Test Sets," *IEEE International Test Conference Proceedings 1986*. 368-374. Washington, D.C.: IEEE Computer Society Press, 1986.

[Hunti 04] Huntington, E.V. "Sets of Independent Postulates for the Algebra of Logic." *Transactions of the American Mathematical Society*, 5: 288-309 (1904).

[IEEE 88] Institute of Electrical and Electronic Engineers, Computer Society Standards Committee. *IEEE Standard VHDL Language Reference Manual*. ANSI/IEEE Std 1076-1987. New York: Institute of Electrical and Electronic Engineers, 1987.

[Igara 79] Igarashi, Yoshihide. "An Improved Lower Bound on the Maximum Number of Prime Implicants," *The Transactions of the IECE of Japan*, E-62: 389-394 (June 1979).

[Inter 87] Intermetrics, Inc. *VHDL Simplifier User's Manual*. U.S. Air Force Contract F33615-83-C-1003. Bethesda, Maryland: 30 April 1987.

[Jacob 87] Jacob, James and Nripendra N. Biswas. "GTBD Faults and Lower Bounds on Multiple Fault Coverage of Single Fault Test Sets," *IEEE International Test Conference Proceedings 1987*. 849-855. Washington, D.C.: IEEE Computer Society Press, 1987.

[Jha 86] Jha, Niraj K. "Detecting Multiple Faults in CMOS Circuits," *IEEE International Test Conference Proceedings 1986*. 514-519. Washington, D.C.: IEEE Computer Society Press, 1986.

[Johns 87] Johnson, E.L and M.A.Karim. *Digital Design: A Pragmatic Approach*. Boston: PWS Engineering, 1987.

[Kirkl 88] Kirkland, Tom and M. Ray Mercer. "Algorithms for Automatic Test Pattern Generation," *IEEE Design and Test of Computers*, 5: 43-55 (June 1988).

[Kohav 72] Kohavi, Igal and Zvi Kohavi. "Detection of Multiple Faults in Combinational Logic Networks," *IEEE Transactions on Computers*, C-21: 556-568 (June 1972).

[Lala 85] Lala, Parag K. *Fault Tolerant and Fault Testable Hardware Design*. Englewood Cliffs: Prentice-Hall International, 1985.

[Lipsc 76] Lipschutz, Seymour. *Discrete Mathematics*. New York: McGraw-Hill, 1976.

[May 84] May, Alan Douglas. *Adaptive Location of Multiple Faults in Combinational Circuits*, MS Thesis. University of Kentucky, Lexington, Kentucky, August 1984.

[McClu 71] McCluskey, Edward J. and Frederick W. Clegg. "Fault Equivalence in Combinational Logic Networks," *IEEE Transactions on Computers*, C-20: 1286-1293 (November 1971).

[Mei 74] Mei, Kenyon C.Y. "Bridging and Stuck-At Faults," *IEEE Transactions on Computers*, C-23: 720-727 (July 1974).

[Mitch 83] Mitchell, O.H. "On a New Algebra of Logic," *Studies in Logic*, edited by C.S.Pierce. Boston: Little, Brown, 1883.

[Nagle 75] Nagle, H.Troy Jr., B.D. Carroll, and J.David Irwin. *An Introduction to Computer Logic*, Englewood Cliffs: Prentice-Hall, 1975.

[Paige 69] Paige, Michael R. *Generation of Diagnostic Tests Using Prime Implicants*. CSL Report R-414. Urbana, Illinois: University of Illinois, 1969.

[Poage 63] Poage, J.F. "Derivation of Optimum Tests to Detect Faults in Combinational Circuits," *Proceedings of the Symposium on Mathematical Theory of Automata, XII*. 483-528 Brooklyn, NY: Polytechnic Press, 1963.

[Quine 52] Quine, W.V. "The Problem of Simplifying Truth Functions," *American Mathematical Monthly*, 59: 521-531 (October 1952).

[Press 87] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 1987.

[Quine 55] Quine, W.V. "A Way to Simplify Truth Functions," *American Mathematical Monthly*, 62: 627-631 (November 1955).

[Rai 78] Rai, Suresh and K.K. Aggarwal. "An Algorithm for Multiple Fault Detection in Combinational Logic Networks," *International Journal of Electronics*, 44: 647-652 (June 1978).

[Rai 87] Rai, Suresh and Rajiv Jain. "A Computer-Aided Technique for Fault Detection in Combinational Circuits," *Microelectronics and Reliability*, 27: 263-265 (1987).

[Rajski 87] Rajski, Janusz and Henry Cox. "A Method of Test Generation and Fault Diagnosis in Very Large Combinational Circuits," *IEEE International Test Conference Proceedings 1987*. 932-943. Washington, D.C.: IEEE Computer Society Press, 1987.

[Reddy 84] Reddy, Sudhakar M., Vishwani D. Agrawal, and Sunil K. Jain. "A Gate Level Model for CMOS Combinational Logic Circuits with Application to Fault Detection," *Proceedings of the Twenty-First Design Automation Conference*. 504-509. Washington, D.C.: IEEE Computer Society Press, 1984.

[Rees 86] Rees, Jonathan and others. "Revised³ Report on the Algorithmic Language Scheme," *ACM Sigplan Notices*, 21: 37-79 (December 1986).

[Reiter 87] Reiter, Raymond. "A Theory of Diagnosis from First Principles," *Artificial Intelligence*, 32: 57-95 (April 1987).

[Roth 84] Roth, J.Paul, Vojin G. Oklobdzija, and John F. Beitem, "Test Generation for FET Switching Circuit," *IEEE International Test Conference Proceedings 1984*. 59-62. Washington, D.C.: IEEE Computer Society Press, 1984.

[Roy 74] Roy, Bhakta K. "Diagnosis and Fault Equivalence in Combinational Circuits," *IEEE Transactions on Computers*, C-23: 955-963 (September 1974).

[Rudeanu 74] Rudeanu, Sergiu. *Boolean Functions and Equations*. Amsterdam: North Holland, 1974.

[Samso 54] Samson, E.W. and B.E. Mills. *Circuit Minimization: Algebra and Algorithms for New Boolean Canonical Expressions*. AFCRC TR 54-21. Cambridge, Massachusetts: Air Force Cambridge Research Center, 1954.

[Scher 72] Schertz, Donald R. and Gernot Metze. "A New Representation for Faults in Combinational Digital Circuits," *IEEE Transactions on Computers*, C-21: 858-866 (August 1972).

[Solana 86] Solana, J.M., J.A. Michell, and S. Bracho. "Elimination Algorithm: A Method For Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis," *IEE Proceedings, Pt. E*, 133: 31-44 (January 1986).

[Stanat 77] Stanat, Donald F. and David F. McAllister. *Discrete Mathematics in Computer Science*. Englewood Cliffs: Prentice-Hall, 1977.

[Tanim 87] Tanimoto, Steven L. *The Elements of Artificial Intelligence: An Introduction Using LISP*. Rockville, Maryland: Computer Science Press, 1987.

[Texas 87a] Texas Instruments, Inc. *PC Scheme: A Simple, Modern LISP User's Guide*. Austin, Texas: July 1987.

[Texas 87b] Texas Instruments, Inc. *TI Scheme: Language Reference Manual*. Austin, Texas: July 1987.

[Turne 85] Turner, Mark E. and others. "Testing CMOS VLSI: Tools, Concepts, and Experimental Results," *IEEE International Test Conference Proceedings 1985*. 322-328. Washington, D.C.: IEEE Computer Society Press, 1985.

[Walcz 84] Walczak, K. and K. Sapiecha. "Multiple Fault Detection and Location in Large Combinational Circuits," *Proceedings of the Fourteenth International Symposium on Fault Tolerant Computing*. 134-140. Washington, D.C.: IEEE Computer Society Press, 1984.

[Wang 75] Wang, David T. "An Algorithm for the Generation of Test Sets for Combinational Logic Networks," *IEEE Transactions on Computers*, C-24: 742-746 (July 1975).

[Yau 71] Yau, Stephen S. and Yu-Shan Tang. "An Efficient Algorithm for Generating Complete Test Sets for Combinational Logic Circuits," *IEEE Transactions on Computers*, C-20: 1245-1251 (November 1971).

[Zasio 85] Zasio, John J. "Non-Stuck Fault Testing of CMOS VLSI," *Proceedings of the Eighth International Computer Society Conference Spring*. 388-391. Washington, D.C.: IEEE Computer Society Press, 1985.

Vita

Captain James J. Kainec was born on [REDACTED]. Following graduation from high school [REDACTED], he received an appointment to the United States Military Academy at West Point, New York. Upon graduation from West Point in May 1982 with a degree of Bachelor of Science, he received a commission as a Second Lieutenant in the United States Army Signal Corps. After completion of the Signal Officer Basic Course at Fort Gordon, Georgia, Captain Kainec was assigned to the 25th Infantry Division, Schofield Barracks, Hawaii. There he served as the Communications Platoon Leader for Headquarters, 1st Infantry Brigade, the Communications-Electronics Staff Officer for the 1st Battalion, 27th Infantry "Wolfhounds", and the Battalion Maintenance Officer for the 125th Signal Battalion. In 1986, Captain Kainec attended the Signal Officer Advanced Course at Fort Gordon, Georgia. Before beginning a graduate program, he completed the US Army Teleprocessing Operations Officer Course at the Air Force Institute of Technology. Following graduation, Captain Kainec will remain at the Air Force Institute of Technology to pursue the degree of Doctor of Philosophy in electrical engineering.

[REDACTED]

~~UNCLASSIFIED~~

SECURITY CLASSIFICATION OF THIS PAGE

Form Approved
OMB No. 0704-0188

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/88D-16		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (if applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6583		7b. ADDRESS (City, State, and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFWAL Microelectronics Lab	8b. OFFICE SYMBOL (if applicable) AFWAL/ELED	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, Ohio 45433		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A DIAGNOSTIC SYSTEM USING BOOLEAN REASONING (UNCLASSIFIED)			
12. PERSONAL AUTHOR(S) James J. Kainec, Captain, US Army			
13a. TYPE OF REPORT MS Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1988 December 5	15. PAGE COUNT 369
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Boolean Algebra, Computer Aided Diagnosis, VHDL Artificial Intelligence, Logic Circuits	
FIELD 09	GROUP 01		
12	02		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Dr. Frank M. Brown, Professor of Electrical Engineering (see reverse)			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Frank M. Brown, Professor		22b. TELEPHONE (Include Area Code) (513) 255-3576	22c. OFFICE SYMBOL AFIT/ENG

for review and distribution
10 Jan 89

The goal of this thesis is to design and implement a diagnostic system for combinational circuits, a type of circuit used in the design of all computers. The diagnostic system is to accept a description of the circuit, supervise an adaptive input-output experiment on a potentially faulty implementation of the circuit, and return the locations of all faults in the circuit.

The description of the circuit which will be input to this system can be in one of two forms: Boolean equations, or statements in the VHSIC Hardware Description Language (VHDL). Based on the circuit structure, a fault model is developed which can be used to mathematically model the state of faults in the circuit. The circuit description is processed to derive a single Boolean characteristic equation; information gained from testing is used to update this equation. The characteristic equation is manipulated to generate test vectors which are used as inputs to the actual circuit being diagnosed. After a given test vector has been input to the circuit, the output is observed. The state of the circuit output is then input to the diagnostic system which uses it to derive new knowledge about the actual circuit. Such tests are conducted repetitively until the diagnostic system determines that further information cannot be derived from testing. At this point, the diagnostic system determines the nature and location of faults in the actual circuit as well as the function actually performed by the circuit.

The basis of the proposed system is Boolean algebraic reasoning. Such reasoning is performed in terms of symbols rather than numbers; thus, a conventional programming language is unsuited for this application. Symbolic programming languages such as LISP or PROLOG, however, handle the manipulation of symbols extremely well. The new diagnostic system is implemented in SCHEME, a modern, efficient dialect of the LISP programming language.